FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: Master's in Computer Science	Spring semester, 2016 Open
Author: Eric Scott Freeman	(signature author)
Instructor: Hein Meling Supervisors: Hein Meling, Leander Jehl, Tormod Lea	
Title of Master's Thesis: Fast and Reliable Byzantine Fault Tolerance	
ECTS: 30	
Subject headings: Distributed Systems Byzantine Faults Publish/Subscribe Pattern Fault Tolerance	Pages: 81 + attachments/other: code (7z file) Stavanger, 15 June 2016

Abstract

Byzantine faults, or arbitrary faults, are difficult to handle due to their unknown nature. They include software errors, hardware errors, and malicious behavior. There are several algorithms which handle Byzantine faults with varying degrees of reliability and speed. The purpose of this thesis is to combine different Byzantine fault tolerant algorithms in a publisher/subscriber service in such a way that achieves high speed and reliability.

Faster algorithms are used for the majority of the publications, but after every α publications, a history publication is sent with a more reliable algorithm. The history publication will allow subscribers to learn any missed publications. Three algorithms are used: Authenticated Broadcast, Bracha's Reliable Broadcast, and Chain. Bracha's Reliable Broadcast is the most reliable of the three, and it is used in broadcasting the history publications.

By combining these algorithms through the use of the history publication, this thesis demonstrates that it is possible to have a distributed, Byzantine fault tolerant service that is both fast and reliable.

Acknowledgments

The author would like to thank Hein Meling for having weekly meetings and for clearing up roadblocks encountered during the thesis. The author would also like to give thanks to Leander Jehl for describing the algorithms and to Tormod Lea for explaining Gorums.

Contents

Abstract **i**

Acknowledgments **ii**

1 Introduction **1**

2 Background **3**

2.1 Byzantine Faults 3
2.2 Quorums 5
2.3 Paxos 5
2.4 Message Authentication Codes 6
2.5 Pub/Sub 6
2.6 Go, Protobufs, and gRPC 8
2.7 Gorums 8

3 Algorithms **10**

3.1 Authenticated Broadcast 10
3.2 Bracha's Reliable Broadcast 13
3.3 Chain 16

4 Related work **20**

4.1 Towards Byzantine Fault Tolerant Publish/Subscribe: A State Machine Approach 20
4.2 Tolerating Arbitrary Failures in a Pub/Sub System 21

5 Design **22**

5.1 Choosing Not to Use Gorums 22
5.2 gRPC 23
5.3 Opening Connections 24
5.4 Subscriber Handling 25

5.5 Publisher Handling . . . 27
5.6 MACs . . . 28
5.7 Coding the Algorithms . . . 30
5.8 History . . . 32
5.9 Unit Testing . . . 32
5.10 Performance Testing . . . 33

6 Implementation **34**
6.1 Protocol Buffers and gRPC Definitions . . . 34
6.2 Publish . . . 36
6.3 Message Handling . . . 38
6.4 Algorithms . . . 38
6.4.1 Authenticated Broadcast . . . 38
6.4.2 Bracha's Reliable Broadcast . . . 40
6.4.3 Chain . . . 41
6.5 Channels and Pointer References . . . 44
6.6 MACs . . . 45
6.7 Malicious Broker . . . 46

7 Results **49**
7.1 Test Setup . . . 49
7.2 Testing Environment . . . 50
7.3 Tests . . . 50
7.3.1 Four Broker Replicas . . . 50
7.3.2 Seven Broker Replicas . . . 58
7.3.3 Multiple Topics . . . 62
7.3.4 Malicious Broker . . . 64

8 Future work **69**
8.1 Optimal α Value . . . 69
8.2 Bottleneck in Chain Publications . . . 69
8.3 Additional Algorithms . . . 70
8.4 Tree Structure . . . 70

9 Conclusion **71**

Bibliography **73**

A Attachments **76**

Chapter 1

Introduction

As people become increasingly more dependent on computers, it is vital that services are available and work correctly. Handling arbitrary faults, or Byzantine faults, in a distributed environment is complex and costly, since the nature of the error is not known, and the nodes in the system need to reach some type of consensus on what to do. It is important that the faults are handled so users can have a safe and reliable service. Nodes can fail due to hardware or software failures. Networks can go down. Also attackers may try to abuse a system.

There is already much work on Byzantine Fault Tolerance in the traditional client-server pattern [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], but there is little in a publish/subscribe pattern. The objective of this thesis is to implement different fault tolerant algorithms in a publish/subscribe pattern. Then performance measurements, such as latency and throughput, will be gathered on algorithms individually and also combined using different α values, where the α value represents how often to use a specific algorithm. Jehl and Meling described the idea of combining these algorithms in [11], but they did not implement it.

The algorithms are used in a publish/subscribe service acting as middleware between publishers and subscribers. This type of service forwards messages from publishers to subscribers that are interested in specific types of messages. An emergency services application is a good example. If there is a large traffic accident, then the police, fire, and medical services would all want to get a notification about the accident. They would be subscribed to this type of notification. If someone has a heart attack, only the medical service would receive the notification. The police and fire departments would not. These are emergency scenarios, and it is vital that the appropri-

ate subscribers get messages. By using Byzantine fault tolerant algorithms in a replicated state machine, the possibility of lost or altered messages is greatly reduced.

Three different algorithms are used in this thesis: Authenticated Broadcast, Bracha's Reliable Broadcast, and Chain. In Authenticated Broadcast, a publisher broadcasts a publication to a set of replicated state machines, which make up a publish/subscribe service. They are also called brokers. The brokers then broadcast the publication to a set of subscribers. In Bracha's Reliable Broadcast, a publisher broadcasts a publication to a set of brokers. Then they broadcast the publication twice: first to themselves and then to themselves and a set of subscribers. In Chain, a publisher sends a publication to only one broker, which then sends the publication to the next broker. Finally the last broker sends the publication to a set of subscribers.

Chapter 2 provides the background information needed to understand the thesis, and Chapter 3 lists the different algorithms used in the thesis. Chapter 4 describes some previous work done at the University of Stavanger concerning Byzantine fault tolerance. In Chapter 5, the design on the application is presented, while the implementation is presented in Chapter 6. Chapter 7 shows the results of testing. Possible future work is mentioned in Chapter 8, and Chapter 9 concludes the thesis.

Chapter 2

Background

The background chapter describes the important terminology, concepts, and libraries that are needed in this thesis. First Byzantine faults, quorums, Paxos, and message authentication codes are discussed in Sections 2.1-2.4. Section 2.5 describes the publish/subscribe paradigm. Then in Section 2.6, there is an overview of the Go language and some Google libraries. Finally in Section 2.7, there is an introduction to gorums, a remote procedure call library used specifically for finding quorums.

2.1 Byzantine Faults

This section describes Byzantine faults, or arbitrary faults, which are a generic group of faults. It is an all-encompassing collection of software, hardware, and network errors as well as malicious attacks. Coding for a specific fault usually just fixes that particular fault, while coding for Byzantine faults tries to account for unexpected faults. Handling Byzantine faults is a major theme in this thesis.

Making a system Byzantine fault tolerate (BFT) is very useful in distributed systems, where one or more nodes could act unexpectedly for a number of reasons. Lamport et al. illustrate this in the Byzantine Generals Problem, which was first described in [12]. This is a non-technical illustration of a Byzantine fault. In the Byzantine Generals Problem, there is a set of generals and their army divisions surrounding an enemy city. They will vote on whether to attack the city or to retreat. They will be successful if they all work together, but they will fail if they do not cooperate. They send messages to each other which carries their vote. The problem occurs when there are one or more traitorous generals. They can send conflicting

messages to the others.

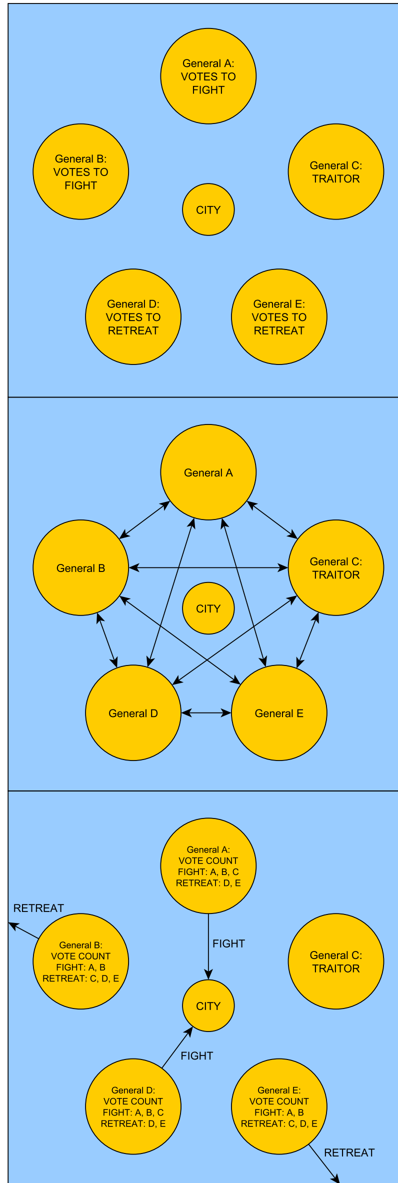

Figure 2.1: Byzantine Generals Problem.

For example, consider Figure 2.1. There are five generals surrounding a

city. Four of them are loyal and cast their votes: two to retreat and two to fight. The fifth general, General C, is subversive however and sends out votes with different contents. Two of the generals will then have a majority of votes to retreat and two will have a majority of votes to fight. It does not really matter what the traitorous general decides to do. The army has failed in this scenario.

It should be reiterated that Byzantine faults are not necessarily malicious. Attacks are a subgroup of Byzantine faults, and Byzantine fault tolerant algorithms are intended to protect against them all, as long as a certain number of the nodes remain correct.

2.2 Quorums

This section describes quorums, which is the minimum number of correct nodes needed to come to an agreement. Quorums are very useful in distributed systems because the nodes in the system need to reach some sort of agreement on what to do. They need to perform the same actions on the same data in the same order. When several nodes share all the same data, they form a replicated state machine. If the nodes in a replicated state machine do not have the same data or perform different operations on their data, their states would deviate from each other. Therefore they need to have a quorum to decide what to do. For example, if someone withdrew money from their bank account in New York City and later checked their account balance a few days later in London, the balance in London could possibly show an incorrect amount if the different replicas in the bank's distributed system never reached an agreement. In typical distributed systems, this would be a majority of the nodes, or $\lceil \frac{n+1}{2} \rceil$ with n representing the total number of nodes. In BFT systems, a quorum is $\lceil \frac{n+f+1}{2} \rceil$ with f representing the tolerable number of faulty nodes [14].

2.3 Paxos

This section gives a brief description of Paxos, which is an early, well-known algorithm using a replicated state machine. Paxos was first proposed by Lamport in [15]. In Paxos, there are three roles which help the algorithm reach consensus: proposers, acceptors, and learners. Each node in the replicated state machine plays each of the three roles. First a client sends a message to a proposer. The proposer will then send *Propose* messages to all the acceptors. If an acceptor supports that proposer as the leader, it will

then send a *Promise* message back. The proposer will then send out *Accept* messages with the client's message to all the accepters if the proposer gets a quorum of *Promises*. Next, each acceptor sends *Learn* messages to all the learners. When a learner gets a quorum of *Learns*, it proceeds to process the message.

2.4 Message Authentication Codes

This section describes Message Authentication Codes (MACs), which are a way of verifying the contents and sender of a message. MACs are very important in this thesis because every network message sent will be verified. Stallings lists four attacks that MACs protect against [13].

1. An attacker that pretends to be someone else. Since two parties share the same key, it is still possible for one party to send itself a message and claim that the other sent it.
2. An attacker that changes the contents of a message.
3. An attacker that reorders the sequence of messages.
4. An attacker that delays messages.

For Message Authentication Codes to work, both parties need a shared, or private, key. The sender calculates the MAC by inputing the message and the key into the MAC algorithm. Then the sender appends the MAC to the message and sends it to the receiver. The receiver then calculates the MAC from their copy of the key and the received message. If the newly created MAC matches the one received from the sender, the message contents and the sender have been authenticated. Figure 2.2 shows the process of adding and verifying a MAC.

MAC algorithms output a fixed length MAC despite receiving a variable length message. This creates a small chance of different inputs creating the same outputs, which is known as a collision. Generally MAC algorithms are designed such that the chance of collisions is small.

2.5 Pub/Sub

A publish/subscribe service is a service in which publishers send publications to the service, and then the service forwards the publications to only the

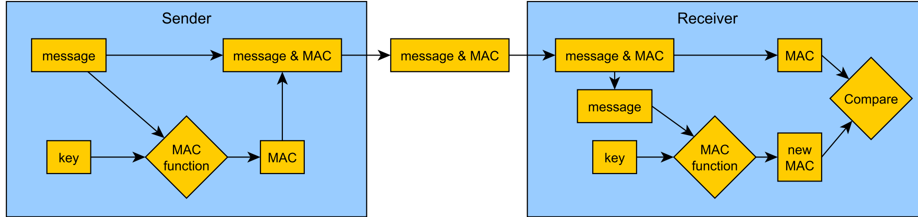

Figure 2.2: Message Authentication Code process.

subscribers who are interested in receiving them. The publishers do not need to know anything about the subscribers. The pub/sub service will handle the delivery of the publications to the interested subscribers. To begin receiving publications, a subscriber must send a subscribe request to the service indicating which content the subscriber would like to receive. Figure 2.3 illustrates the basic idea of a publish/subscribe service, which is also called a broker service in this thesis. It is difficult to find the origins of the pub/sub service. It was described as a "news service" by Birman and Joseph in [16] in 1987. Now it is a common pattern.

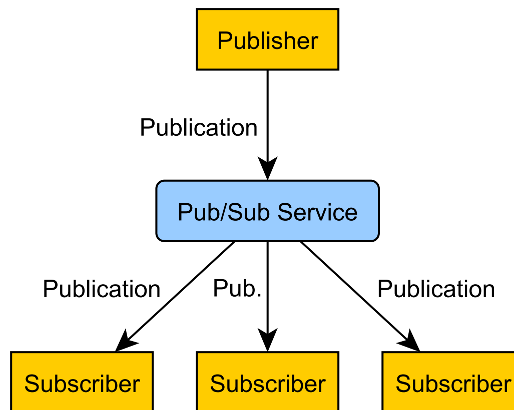

Figure 2.3: Publisher/Subscriber Service.

A good example of a publish/subscribe service would be an alarm service for a company. The other services the company provides would be the publishers. They would send different events which occur to the alarm service.

A manager might want to know whenever an error or a critical event occurs on one of the services, so he/she would subscribe to errors and critical events.

2.6 Go, Protobufs, and gRPC

This section describes a few of Google's technologies that are used in this thesis: Go, protocol buffers, and gRPC.

Go, or golang, is Google's programming language which is designed for back-end programs. All of the code used in this thesis is written in golang. Go has a big focus on concurrency. There is a slogan for Go which states "Do not communicate by sharing memory; instead, share memory by communicating [17]." Usually in programming languages, memory is shared among processes by allowing shared access to a variable that is locked when a process wants to use it. This is allowed in Go, but it is preferred pass values through channels instead. Channels prevent race conditions and allow programmers to write cleaner code [17].

Protocol buffers (protobufs) are Google's way of serializing data. They define messages. They are not tied to a specific language, and they are many times smaller and faster than XML since they are encoded as binary instead of text [18]. Protobufs were also designed to be simpler to use programmatically than XML is to use.

Google has also created a remote procedure call library called gRPC. Remote procedure calls allow a process to run functions on a different machine, which simplifies communication between machines. gRPC is used to define services which contain remote procedure calls. It primarily uses protobufs to define them, but can also use other types of serialization [19]. The network communication in this thesis uses gRPC. Both protobufs and gRPC are open-source.

2.7 Gorums

This section describes Gorums [20], which is a remote procedure call library written by Tormod Lea to simplify finding quorums in replicated state machines. It is a portmanteau of Go and quorums. It uses both gRPC and protocol buffers. Through gorums, users can specify what constitutes a quorum for every remote procedure call. Gorums also generates a manager, which manages the configurations of the distributed system.

Gorums has three types of remote procedure calls [21]. First there is the basic synchronous call, where the client will send one request to each

server and will block until a quorum of replies are received. Next there is the asynchronous call. Here the client sends one request to each server, but it does not block while it waits for the replies. Finally there is one-way client streaming. Here the client can send multiple messages to the servers. It just broadcasts messages to all the servers without ever checking for a quorum. There is no one-way server streaming or two-way streaming in gorums.

Chapter 3

Algorithms

This chapter discusses the algorithms needed for this thesis. The first is the Authenticated Broadcast algorithm which is used to verify the source and contents of a message by using several replicas forward the message to the destinations. The next algorithm is Bracha's Reliable Broadcast, which requires the replicas to echo the message to each other to reach a consensus before sending the message to the destinations. The third algorithm is Chain where a message is passed down a chain of replicas to eventually reach the destination.

3.1 Authenticated Broadcast

This section describes the Authenticated Broadcast (AB) algorithm described by Srikanth and Toueg in [22]. Cachin et al. [14] offer an easier to follow explanation of the same algorithm, which they call Authenticated Echo Broadcast. In the AB algorithm, a source sends a message to all the replicas in the system. A total of $3f + 1$ replicas are needed. The replicas then forward the message to the destination. When the destination has received enough ($2f + 1$) correct messages, the original message is considered to be authenticated, and the destination can proceed to process it.

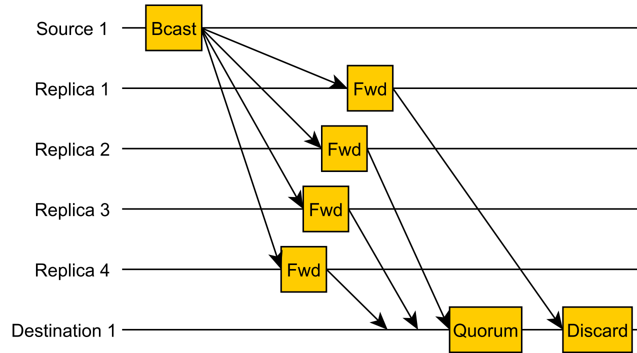

Figure 3.1: Authenticated Broadcast.

Figure 3.1 shows an example execution of the algorithm. This example has a fault tolerance of one. The source sends a message to all the replicas, which receive it at different times. The replicas then forward the message to the destination. When the destination receives three correct messages, which is enough to form a quorum, it proceeds to process it. The fourth message received is no longer needed, and it is discarded.

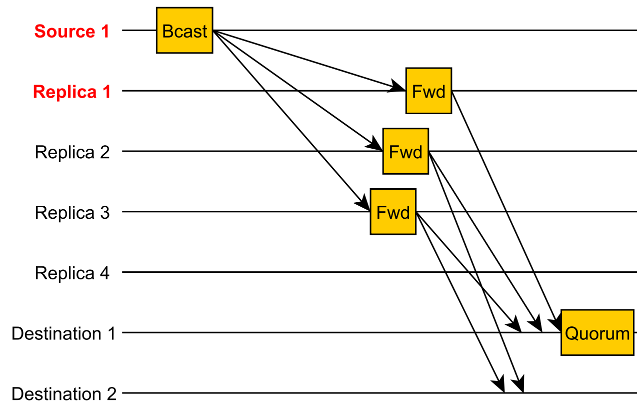

Figure 3.2: Authenticated Broadcast Error Scenario.

If both the sender and one or more replicas are faulty, it is possible for one destination to receive enough messages to reach consensus, while another destination does not receive enough. Figure 3.2 shows an example of this. Source 1 and Replica 1 are faulty. Source 1 sends the message to

all the replicas except Replica 4. Then Replica 1 forwards the message to Destination 1 but not Destination 2. The other replicas behave correctly. Destination 1 receives enough of the messages to form a quorum and thus reach agreement, but Destination 2 does not receive a quorum necessary to reach consensus.

Algorithms 1 - 3 show the pseudocode for Authenticated Broadcast. To simplify the pseudocode, we will assume only one message is ever sent from the source. More messages could be sent later with the inclusion of message IDs.

Algorithm 1 Authenticated Broadcast: Source

```
1: function OnEvent(Send, m)
2:     for all replicas do
3:         Send⟨m⟩
```

Algorithm 2 Authenticated Broadcast: Replica

```
1: function Init
2:     sent ← false
3:
4: function OnEvent(Receive, m)
5:     if sent = false then
6:         for all destinations do
7:             Send⟨r, m⟩
8:         sent ← true
```

Algorithm 3 Authenticated Broadcast: Destination

```
1: function Init
2:     messages ← ∅
3:     processed ← false
4:
5: function OnEvent(Receive, r, m)
6:     if tuple (r, m) ∉ messages then
7:         messages ← messages ∪ tuple (r, m)
8:         if #messages > 2f & processed = false then
9:             Process(m)
10:            processed ← true
```

3.2 Bracha's Reliable Broadcast

This section discusses Bracha's Reliable Broadcast algorithm (BRB) explained in [23]. Cachin et al. [14] offer an easier to follow explanation of the algorithm, which they call Authenticated Double-Echo Broadcast. In addition to authenticating the source and content, the BRB algorithm ensures that all correct replicas eventually process the message from the source.

In Bracha's Reliable Broadcast algorithm, the source sends a message to all replicas. When the replicas receive the message, they echo it to all the replicas, including themselves. When the replicas have received a quorum of correct echoes, $\lceil \frac{n+f+1}{2} \rceil$, they send ready messages to all the replicas and the destinations. When a destination receives a consensus of correct ready messages, $2f+1$, it processes the message. Also included in BRB is a safety net which ensures the totality of all correct replicas process the message from the source. If a replica receives $f+1$ ready messages, it will then send its own ready message to all the replicas and the destinations. While BRB is great at guaranteeing that all correct replicas process the message, there is a lot of communication.

Figure 3.3 shows the ideal execution of the BRB algorithm. To keep the figure from becoming too messy, all timing (processing time and network delays) is shown to be constant. Also the replicas are not shown sending messages to themselves, which can be optimized out.

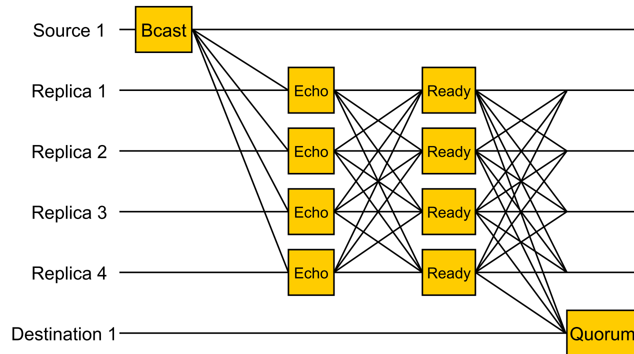

Figure 3.3: Bracha's Reliable Broadcast.

Figure 3.4 shows how BRB ensures totality even with a faulty sender and a faulty replica. The errors in this scenario are similar to the ones shown in Figure 3.2. Source 1 and Replica 1 are faulty and only send messages

selectively. Source 1 sends the message to Replicas 1-3 but not to Replica 4. Therefore Replicas 1-3 send out echo messages, but Replica 4 does not. Since Replica 1 is faulty, it chooses to only send echo messages to Replicas 2-3 and itself. Replicas 1-3 each receive 3 echo messages, while Replica 4 only receives 2. Three echo messages is enough to satisfy the quorum, but two is not. Replicas 1-3 then send out ready messages. Replica 1 does not send any to Replica 4 and Destination though. Now Replica 4 receives two ready messages, which is enough to satisfy the second condition for sending ready messages. It can send out its own ready messages now. Finally Destination 1 receives enough ready messages to process them.

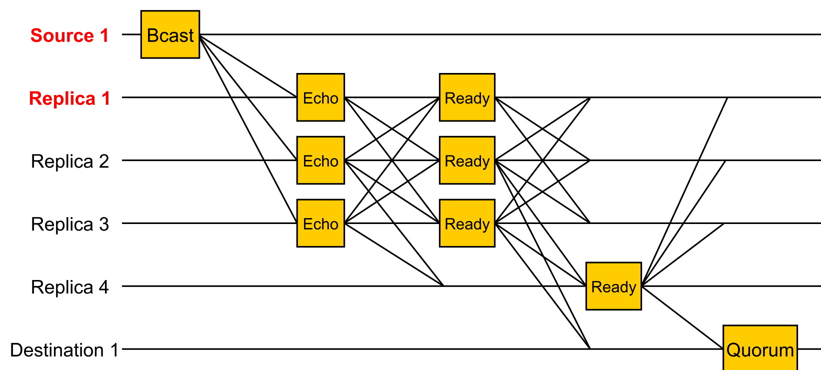

Figure 3.4: Bracha's Reliable Broadcast Error Scenario.

Algorithms 4 - 6 show the pseudocode for Bracha's Reliable Broadcast.

Algorithm 4 Bracha's Reliable Broadcast: Source

1: **function** OnEvent(Send, m)
2: **for all** replicas **do**
3: Send$\langle SEND, m\rangle$

Algorithm 5 Bracha's Reliable Broadcast: Replica

```
1:  function INIT
2:      echoSent ← false
3:      echoes ← ∅
4:      readySent ← false
5:      readys ← ∅
6:
7:  function ONEVENT(Receive, SEND, m)
8:      if echoSent = false then
9:          for all replicas do
10:             Send⟨ECHO, r, m⟩
11:         echoSent ← true
12:
13: function ONEVENT(Receive, ECHO, m)
14:     if tuple (r, m) ∉ echoes then
15:         echoes ← echoes ∪ tuple (r, m)
16:         if #echoes > (n+f)/2 & readySent = false then
17:             for all replicas & destinations do
18:                 Send⟨READY, r, m⟩
19:             readySent ← true
20:
21: function ONEVENT(Receive, READY, m)
22:     if tuple (r, m) ∉ readys then
23:         readys ← readys ∪ tuple (r, m)
24:         if #readys > f & readySent = false then
25:             for all replicas & destinations do
26:                 Send⟨READY, r, m⟩
27:             readySent ← true
```

Algorithm 6 Bracha's Reliable Broadcast: Destination

```
1:  function Init
2:      readys ← ∅
3:      processed ← false
4:
5:  function OnEvent(Receive, READY, r, m)
6:      if tuple (r, m) ∉ readys then
7:          readys ← readys ∪ tuple (r, m)
8:          if #readys > 2f & processed = false then
9:              Process(m)
10:             processed ← true
```

3.3 Chain

Next we review the Chain algorithm, which is described in [1] by Aublin et al. In Chain, a set of $2f + 1$ replicas are chosen to be a chain. A source sends a message to the first replica in the chain along with $f + 1$ MACs, one for each of the next $f + 1$ nodes in the chain. The first replica verifies the MAC of the sender, adds its own $f + 1$ MACs for the next $f + 1$ nodes in the chain, and sends the message to the next node. Each node verifies the MACs for the previous $f + 1$ nodes, if there are at least that many previous nodes, and adds its own MACs. The last replica then sends the message to the destination. If the destination or any of the replicas cannot verify the MACs, the message is discarded.

Figure 3.5 shows an execution of the Chain algorithm without errors. Since only $2f + 1$ replicas are needed, Replica 4 is not used.

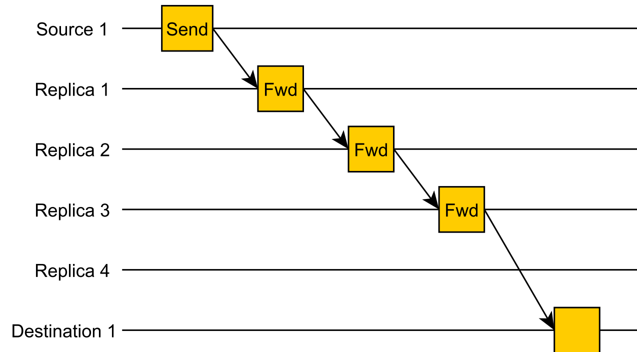

Figure 3.5: Chain.

Figure 3.6 show an execution of the Chain algorithm where Replica 2 is faulty. Replica 2 modifies message $m1$ to $m1'$. When Replica 3 gets message $m1'$, the MAC from Replica 1 no longer matches, and the message is discarded. Unfortunately Destination 1 does not get the message intended for it.

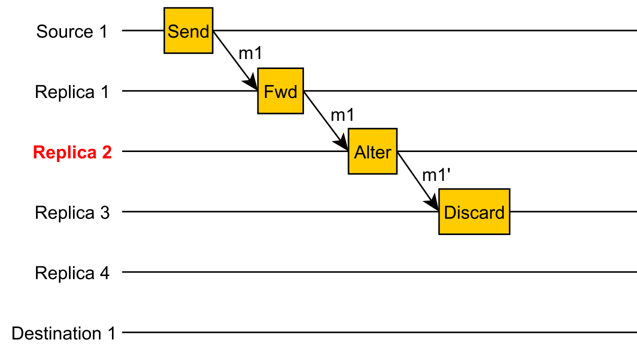

Figure 3.6: Chain Error Scenario.

Algorithms 7 - 9 show the pseudocode for the Chain algorithm.

Algorithm 7 Chain: Source

```
1: function Init
2:     keys ← set of keys for first f + 1 replicas
3:
4: function OnEvent(Send, m)
5:     macs ← ∅
6:     for f + 1 replicas do
7:         macs ← macs ∪ CalculateMAC(keys, m)
8:     Send⟨m, macs⟩
```

Algorithm 8 Chain: Replica

```
1: function Init
2:     sent ← false
3:     keys ← set of keys for prior f + 1 replicas and next f + 1 replicas
4:
5: function OnEvent(Receive, m, macs)
6:     verified ← true
7:     if sent = false then
8:         for f + 1 previous replicas do
9:             if VerifyMAC(m, macs, keys) = false then
10:                verified ← false
11:        if verified = true then
12:            for f + 1 next replicas do
13:                macs ← macs ∪ CalculateMAC(keys, m)
14:            Send⟨m, macs⟩
15:            sent ← true
16:
```

Algorithm 9 Chain: Destination

```
1: function INIT
2:     processed ← false
3:     keys ← set of keys for last f + 1 replicas
4:
5: function ONEVENT(Receive, m, macs)
6:     verified ← true
7:     if processed = false then
8:         for f + 1 previous replicas do
9:             if VerifyMAC(m, macs, keys) = false then
10:                verified ← false
11:        if verified = true then
12:            Process(m)
13:            processed ← true
14:
```

Chapter 4

Related work

This chapter describes some of the previous projects by professors and students at the University of Stavanger. This thesis is a continuation of their work. Section 4.1 describes the paper by Jehl and Meling. This thesis implements the designs described by them. Section 4.2 describes an earlier thesis which tested the throughput of Authenticated Broadcast and Bracha's Reliable Broadcast.

4.1 Towards Byzantine Fault Tolerant Publish/Subscribe: A State Machine Approach

This section reviews [11], in which Jehl and Meling discuss ways of combining the algorithms mentioned before in a publisher/subscriber system as well as how the algorithms can work in a tree structure. While Authenticated Broadcast is cheap in regards to the number of messages it sends, its consensus is weak. Refer to Figure 3.2 as an example. On the other hand Bracha's Reliable Broadcast has strong consensus, but it is expensive regarding the number of messages required to reach consensus. The paper suggests combining the two algorithms. Use AB for the majority of the publications, and then use BRB once after every α messages to guarantee that all correct subscribers get caught up on the publications. The BRB messages would contain a history of the previous α messages. Also mentioned is replacing AB with Chain, since Chain would work better in tree structures [11].

The paper then discusses ideas for a tree structure made up of Byzantine fault tolerant (BFT) brokers to forward the publications to any interested parties, which could be subscribers or other brokers. Figure 4.1 show an example of a BFT broker tree. The brokers are each a replicated state

machine made up of acceptors, a term borrowed from Paxos. It is possible to combine acceptors from different brokers to reduce the number of messages sent. Also mentioned is a way to update the subscriptions throughout the tree. This thesis will just focus on the first part of Jehl and Meling's paper, combining the algorithms.

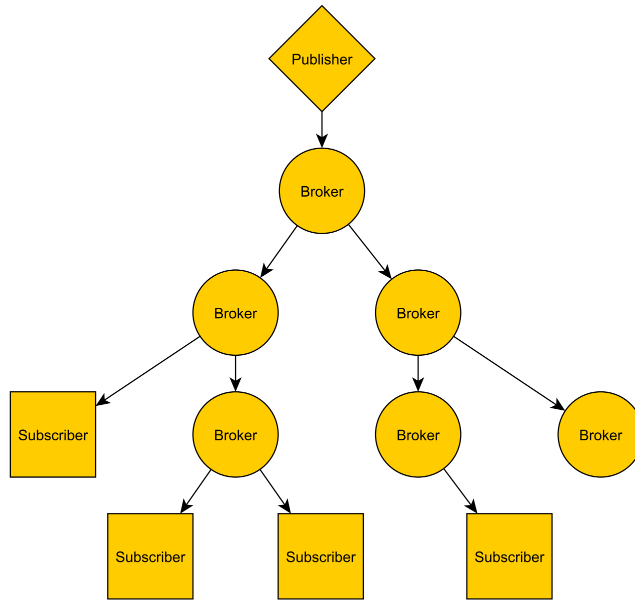

Figure 4.1: BFT Broker Tree.

4.2 Tolerating Arbitrary Failures in a Pub/Sub System

In [24], Haaland and Ramstad started to implement the ideas expressed in Jehl and Meling's paper. They used goxos, a Go implementation of Paxos, to code the algorithms for AB and BRB, but they did not combine the algorithms. They also used Paxos terminology in naming the processes and messages. The performance of the algorithms was tested, and they determined that the throughput of AB is about three times greater than BRB's throughput [24].

Chapter 5

Design

This chapter reviews the design process used in this thesis. Section 5.1 discusses the reason why gorums was not used. Section 5.2 covers how gRPC is used in the thesis. Section 5.3 reviews how the different processes connect to each other. Next, Sections 5.4 and 5.5 discuss how the broker replicas handle connections to subscribers and publishers. Section 5.6 describes how the messages are authenticated, and Section 5.7 provides a general view of how the project is coded. Section 5.8 describes how the history works with different topics from the same publisher. Finally, Sections 5.9 and 5.10 tell how unit testing and performance testing are carried out.

5.1 Choosing Not to Use Gorums

While learning how to use Gorums, it was decided that it is not currently a good fit for a pub/sub system. Asynchronous broadcasting is used heavily in the algorithms. Since gorums does not check for quorums in the one-way client streaming calls, the quorums from these calls would need to be checked manually, i.e. not by Gorums. Using Gorums's quorum checking functionality was one of the major points in wanting to use it. Additionally, in order for the replicas to broadcast publications to the destinations, or subscribers, the replicas would need to be clients of the subscribers. This felt counterintuitive because the subscribers should be clients of the replicas. The replicas are the ones providing the service. Clients are the ones which initiate the connections. Therefore it was decided to use basic gRPC calls for developing the publish/subscribe service. gRPC allows for streaming in either direction. This allows both publishers and subscribers to be clients of the service.

5.2 gRPC

This section covers the communications between the different processes via gRPC. There are three separate gRPC services specified:

- Communication between the publishers and the brokers
- Communication between the subscribers and the brokers
- Communication between the brokers

The gRPC calls in the subscriber service are streaming in both directions, since the brokers (servers) should be able to send publications back to the subscribers (clients) at any time. The subscribers should also be able to send subscription updates at any time.

The calls on the other services are basic request/response calls. In the publisher service, the requests from the publishers (clients) contain the publication, and the responses from the brokers (servers) contain a status code signifying whether the α value has been reached and other things. Recall from Section 4.1 that the α value represents how many AB publications should be sent before a BRB publication. In the inter-broker service, the requests contain the publication, and the responses contain a status code.

Figure 5.1 illustrates the gRPC services. The basic request/response call is shown with a single arrow. Two-way streaming is represented with multiple arrows in both directions.

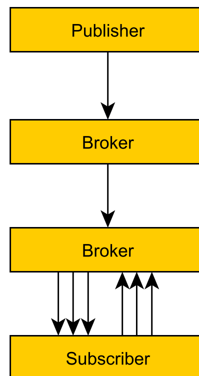

Figure 5.1: gRPC services.

5.3 Opening Connections

This section discusses how the different processes connect to each other. The broker service is a replicated state machine, so it has several replica nodes. For BRB and Chain, the replicas need to communicate with each other, so they try to connect with each other on startup or after failures occur. Communication between replicas is not necessary for AB. Both the publisher and the subscriber are clients of the broker service, so they initiate connections with the broker replica nodes. Figure 5.2 shows in general how the connections are made.

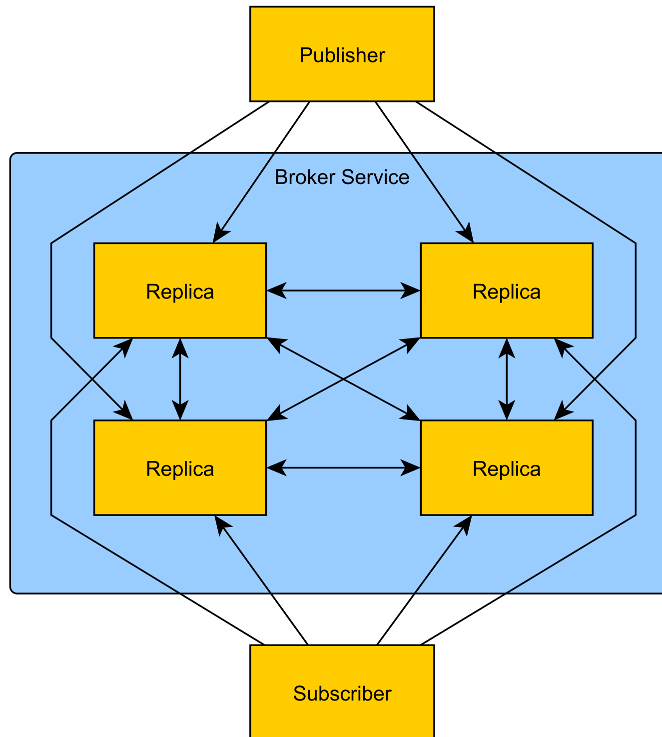

Figure 5.2: Opening connections.

5.4 Subscriber Handling

This section covers the design of communication between a subscriber and a broker replica. When the subscriber is created, a connection is created to each broker replica, and a subscribe call is started. A read loop and a write loop are then started in the subscriber for the streams in that call. The subscriber then sends a subscribe request to a broker replica. When the broker receives the initial message, it records subscriber information like the subscriber's ID and publication filters (topics). The replica then starts a read thread and a write thread for the streams in the subscriber's call. Whenever the broker replica has processed a publication and placed it on the output channel for the subscriber, the write loop sends the publication to the subscriber. This can and most likely will happen multiple times. See Figure 5.3 for an illustration of this. The ellipses represent other processing done by the broker node. The publications can be added to the output channels at any time. Also the subscriber can send new subscribe requests to the broker nodes with updated filters. This is not shown in the figure.

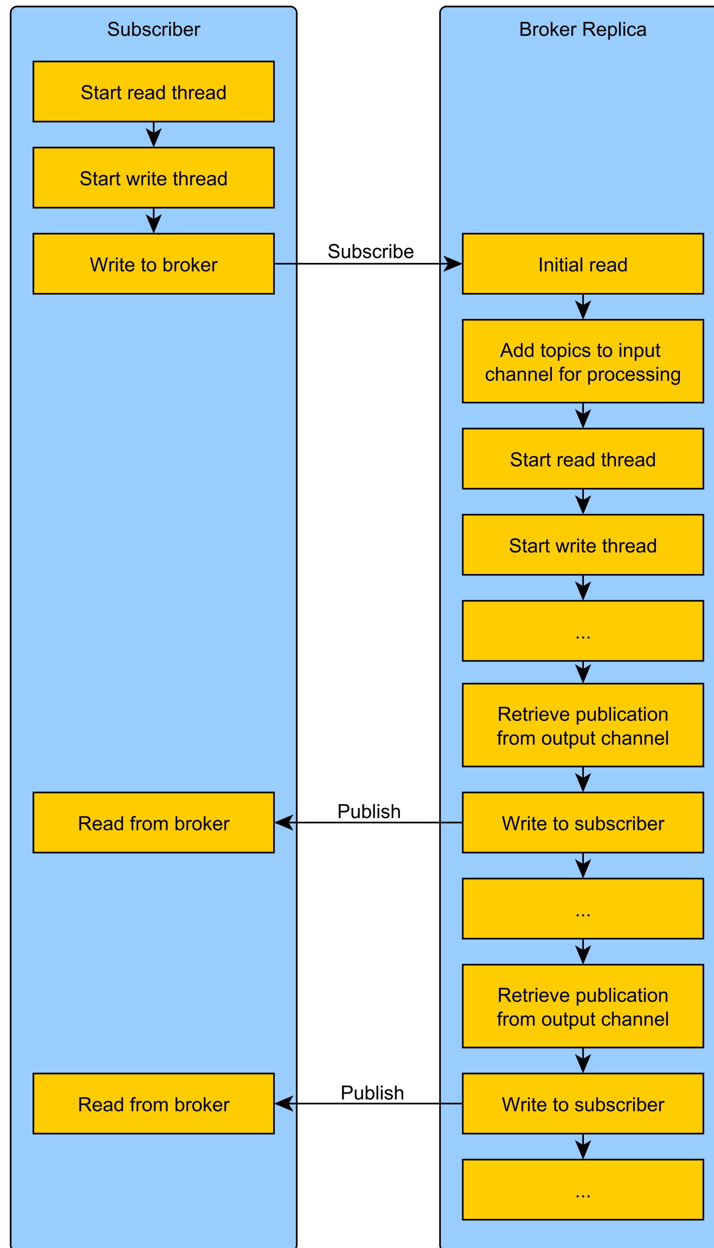

Figure 5.3: Subscriber handling.

5.5 Publisher Handling

This section covers the design of communication between a publisher and a broker replica. This is a little simpler than the communication between a subscriber and a broker replica, because there is no streaming involved between the publisher and the broker replica. When the publisher is created, a connection is created to each broker replica. On this connection, multiple publication calls can be made, each with a synchronous request and response. Therefore there does not need to be any loops to handle the streams. The publisher can send a publish call at anytime and then waits for the brokers' responses, which will contain a status code stating whether or not the α value has been reached. When the replica receives a publish call from the publisher, it places the publication on a channel for processing and return the response to the publisher. See Figure 5.4 for an illustration of this. Note that communication between the brokers is similar to this.

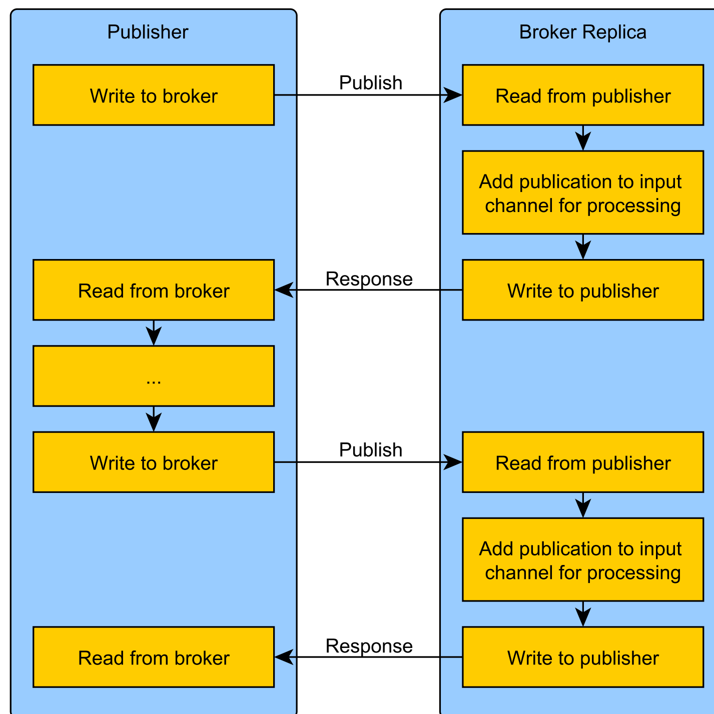

Figure 5.4: Publisher handling.

5.6 MACs

This section describes the design plan for using Message Authentication Codes. gRPC has built-in authentication methods that could have been used in this thesis. It would have simplified the project to use them as well, instead of manually adding and checking MACs. Unfortunately they do not work with the Chain algorithm, which requires multiple MACs to be added to the message. The gRPC authentication only works between the source and destination, not from a chain of sources. It would have been possible to use gRPC's authentication for the source immediately prior to the destination and to manually check the MACs for the other sources, but then any comparisons between the algorithms might be skewed if one authentication method was noticeably faster than the other. Therefore it was decided to manually add and verify MACs.

For the AB and BRB algorithms, MACs are added to the messages immediately before sending to their destinations. All pertinent content needs to be updated beforehand. MACs are verified immediately upon receipt to check that the sender and the content are correct. Messages with invalid MACs are discarded.

Adding MACs for the Chain algorithm is a little more complicated. Let us consider a scenario where $f = 1$. For each received message, a node needs to check $f+1 = 2$ MACs (if there are that many). So at first glance, it might appear that each message needs to include two MACs: one from the parent node to the current node, and one from the grandparent node. In addition to these two MACs, there is a group of MACs that the parent node must add for the current node's children. They also need to check two MACs.

Figure 5.5 shows an example of the MACs that need to be included in the messages. In this scenario, there is one publisher, three broker replicas, and three subscribers. The MACs are named by their from and to locations. So a MAC between Publisher 1 and Broker Replica 1 is named $P1B1$ in the figure. The MACs are colored according to their type: a MAC from a parent node to the current node is colored blue, a MAC from the grandparent to the current node is colored pink, and MACs from the parent to the grandchildren are colored green. For example, let us consider the message between Broker Replica 2 and Broker Replica 3. Broker Replica 3 needs to verify $B1B3$ and $B2B3$. It also needs to include one of the MACs $B2S1$, $B2S2$, or $B2S3$ in the messages it sends to its children. The children also need to verify two MACs.

In a more complete system, there would need to be some type of key distribution system utilized. For the sake of time, the private (shared) keys are

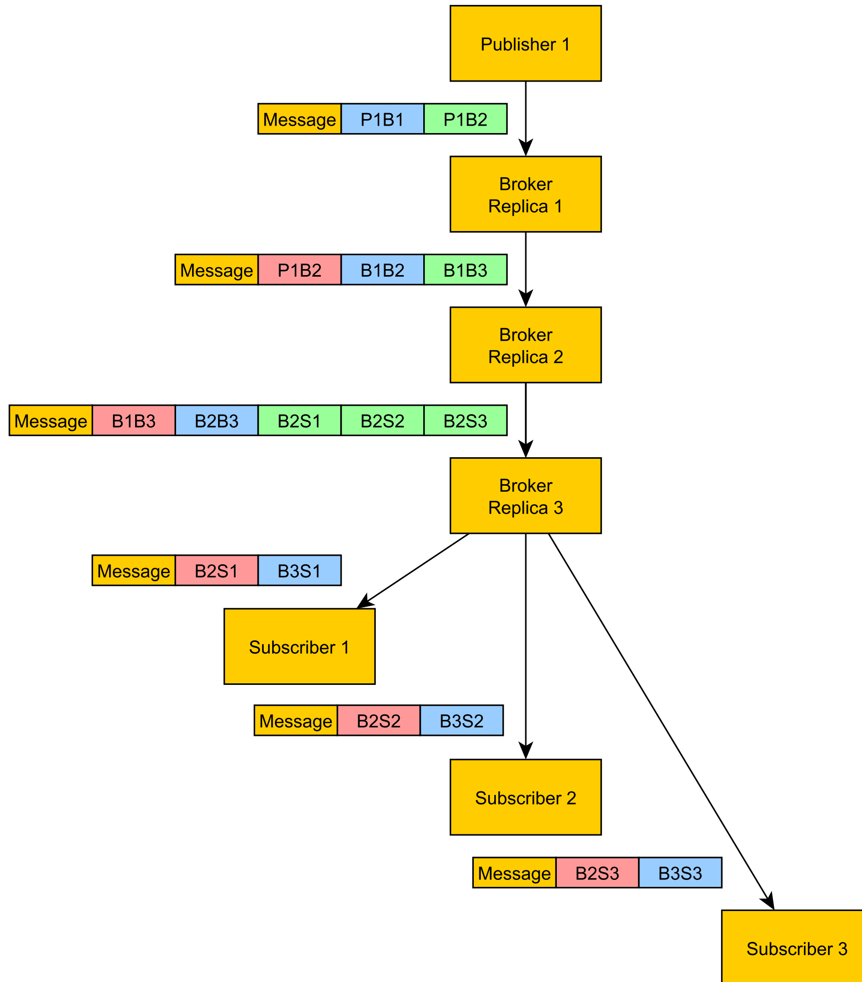

Figure 5.5: Chain MACs.

kept in text files and loaded whenever the brokers, publishers, or subscribers are started. There needs to be $\frac{n \cdot (n-1)}{2}$ keys for communication between n broker replicas. For each publisher and for each subscriber, there needs to be n keys as well. For example, if there are 4 broker replicas, 2 publishers, and 5 subscribers, there needs to be $\frac{4 \cdot (4-1)}{2} + 2 \cdot 4 + 5 \cdot 4 = 34$ keys. The keys are randomly generated.

5.7 Coding the Algorithms

This section describes the plan on how to break up the program into smaller pieces so that writing the code will be more manageable. The algorithms were coded separately at first to verify that were working as expected. Once they were working satisfactorily, they were combined.

The majority of the algorithms are performed in a separate goroutine (thread). By this, I mean that the threads performing network communication are not handling the algorithms. There is a read thread and a write thread for the streaming calls. They continually loop until the connection is terminated. The threads communicate with the handler thread through channels. Whenever a read thread receives a message, the read thread places the message into a channel to be processed by the handler thread. Channels are described in Section 2.6 on Go. Once it has been processed, the handler thread puts a new message into a channel for a write thread. The write thread then forwards the message to its new destinations. Algorithm 10 shows some pseudocode explaining the basic idea of having the read and write loops communicate through channels with streaming calls. The handler thread is started when the process is created.

The basic, non-streaming calls also use channels to communicate with the handler thread. Both the read and write happen in the same thread, and they execute sequentially. This occurs once per call. Algorithm 11 shows the general flow of a basic, non-streaming call.

While most data is shared through channels, there is some shared memory and read/write mutexes. There needed to be a way to keep track of the channels that are created. There are maps of the channels with a connection ID used as a the key. This allows for quick lookup of a channel to a particular node. Additions and deletions to a map are write-locked, while reads are read-locked. Multiple threads can read at the same time, but during a write-lock only one process has access to that memory.

Algorithm 10 Channels in streaming calls.

```
1: function StreamingCall
2:     Begin ReadThread
3:     Begin WriteThread
4:
5: function ReadThread
6:     for ... do
7:         message = connection.Read()
8:         InputChannel ← message
9:
10: function HandlerThread
11:     for ... do
12:         message ← InputChannel
13:         Process message
14:         OutputChannel ← message
15:
16: function WriteThread
17:     for ... do
18:         message ← OutputChannel
19:         connection.Write(message)
```

Algorithm 11 Channels in basic calls.

```
1: function BasicCall(message)
2:     InputChannel ← message
3:
4: function HandlerThread
5:     for ... do
6:         message ← InputChannel
7:         Process message
8:         OutputChannel ← message
9:
```

5.8 History

This section describes how to send the history of the publications after every α are sent. Recall from Section 4.1 that AB is used to send α publications, and then a history of those is then sent via a BRB publication. This provides a better guarantee that all the interested subscribers receive the publications since BRB is more reliable.

It becomes a bit more interesting when filters for the different topics are applied. For example, a publisher could send out publications with a variety to topics. The subscribers should not receive topics that they are not interested in or have permission to see, but it would be possible for the history to contain multiple topics. Two solutions to this issue were considered.

The first solution would be to just use a single α counter for each publisher. The publisher would send out history publications which could possibly contain a number of different topics, and the brokers would split up the history into individual publications after they had received a quorum of ready messages. They would then check the topic for each publication and send it to each subscriber that is subscribed to that topic.

The second solution would be use a different counter for every publisher/topic. So if a publisher has three topics, there would be three counters for that publisher. Whenever that publisher reaches the α value for that topic, it would send a history BRB for that particular topic. Since all the publications in the history share the same topic, the brokers would not need to split up the history. They can just send it to the subscribers who are subscribed to that topic. It was decided to use this second approach.

5.9 Unit Testing

This section describes the unit testing design. It was important to think about this early so that the code did not have to be rewritten in order to be tested. Unit tests were written for individual functions within the broker replica, the publisher, and the subscriber. In the replica nodes, tests were written to make sure the messages were processed correctly by the different algorithms and sent to the correct destinations. The publisher was tested to make sure that it sent a history publication when it had received a quorum of history requests (when the α value is reached) from the broker replicas. The subscriber was tested to verify that it handled both normal publications and history publications correctly. Having unit tests was helpful in determining if any changes during the coding process broke existing functionality.

5.10 Performance Testing

This section describes the design of the performance tests. The main goal of this thesis is determining how efficient the algorithms and the combination of the algorithms are. Therefore it is important to design good performance tests.

There are two separate processes running on the same machine. One is a publisher process, and the other is a subscriber process. The publisher process publishes a certain number of publications and records the times when they were sent, while the subscriber process receives the publications and records the times when they were received. Then the differences in the times are calculated. This gives us the latencies. Throughput is measured by the number of times the last step in an algorithm is preformed in the brokers. By running both the publisher and subscriber on the same machine, this eliminates any possible time synchronization issues. See Figure 5.6.

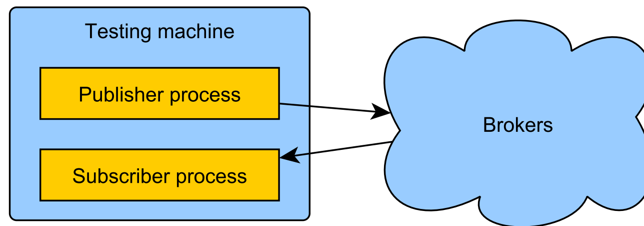

Figure 5.6: Test setup.

In addition to the normal testing scenarios, two additional scenarios are tested. A broker replica node can purposefully misbehave either by omitting or altering a publication. An optional command line option allows a broker node to act erroneously. When the node misbehaves, we are able to determine if the malicious behavior is caught.

Chapter 6

Implementation

This chapters describes how the project was coded. Some sample code is shown as examples. Section 6.1 describes the message and function definitions used in communication between the different processes using protocol buffers and gRPC. Section 6.2 shows the implementation of Publish, the most important gRPC function in this project. In Section 6.3, the function for the message processing thread is shown. Section 6.4 describes some important pieces of code for the different algorithms. Section 6.5 reviews one of the issues encountered while working with channels. Section 6.6 covers adding and checking MACs, and Section 6.7 describes how a replica was made to be malicious for testing purposes.

6.1 Protocol Buffers and gRPC Definitions

This section describes the protobuf message definitions and the gRPC service definitions. Listing 6.1 shows what is contained inside a Publication. It contains the publication type, the ID of the publisher, the ID of the publication, the publication's topic, the ID of the broker which processed it most recently, the contents or message of the publication, and finally one or more message authentication codes. Each variable is assigned a number specifying the order in which they appear in the message.

PubType is an enumeration of the different publication types: Authenticated Broadcast, Bracha's Reliable Broadcast, and Chain. It is defined as $uint32$, which is the smallest unsigned integer type allowed by protobufs. $PublisherID$, $TopicID$, and $BrokerID$ are all 64-bit unsigned integers. While this project will not need anywhere close to 64 bits, IPv4 and Y2K have shown that it is better to err on the side of caution. $PublicationID$ is

slightly different since it is declared as *sint*64. Both *int*64 and *sint*64 define signed 64-bit integers, but protobufs handle negative values more efficiently with *sint*64 [18]. The special history publications sent after every α publications should not interfere with the publishers' numbering scheme. Therefore the history publications will have publication IDs with negative numbers. *Contents* is an array of byte arrays. Use of the keyword *repeated* is used to create an array. It contains the actual messages to be published. Most of the publications will just have one message in *Contents*, but the history publications will contain multiple messages. *MAC* is a byte arrays containing the MAC used between the sender and receiver of a message. All three algorithms use the *MAC* field. *ChainMACs* is an array of the *ChainMAC* message, which contains a MAC, who created the MAC, and who will use the MAC. With *ChainMACs* the Chain algorithm can use the additional MACs described in Section 5.6.

Listing 6.1: Protobuf Publication definition

```
1  message Publication {
2    uint32 PubType = 1;
3    uint64 PublisherID = 2;
4    sint64 PublicationID = 3;
5    uint64 TopicID = 4;
6    uint64 BrokerID = 5;
7    repeated bytes Contents = 6;
8    bytes MAC = 7;
9    repeated ChainMAC ChainMACs = 8;
10 }
11
12 message ChainMAC {
13   string From = 1;
14   string To = 2;
15   bytes MAC = 3;
16 }
```

Three separate services were defined: one for publisher-broker communication, one for subscriber-broker communication, and one for inter-broker communication. They were split up to help prevent a publisher or subscriber from accidentally calling a remote procedure call (RPC) which they should not use. The use of MACs would prevent the RPCs from processing even if they were called. Both the *PubBroker* service and the *InterBroker* service use basic request/response calls, but the *SubBroker* service uses streaming calls. The *SubBroker* service needs to keep the call open so that the brokers (servers) can send multiple asynchronous messages back to the subscribers (clients). Listing 6.2 shows the gRPC service definitions.

Listing 6.2: GRPC Publish and Subscribe definitions

```
1 service PubBroker {
2   rpc Publish(Publication) returns (PubResponse) {}
3 }
4
5 service SubBroker {
6   rpc Subscribe(stream SubRequest) returns (stream Publication) {}
7 }
8
9 service InterBroker {
10   rpc Echo(Publication) returns (EchoResponse) {}
11   rpc Ready(Publication) returns (ReadyResponse) {}
12   rpc Chain(Publication) returns (ChainResponse) {}
13 }
```

6.2 Publish

This section describes the implementation of the *Publish* gRPC function on the broker. This function is called by a publisher whenever it wants to publish something. Listing 6.3 shows the code.

Listing 6.3: Publish

```
1 func (b *Broker) Publish(ctx context.Context, pub *pb.Publication) (*pb.
      PubResponse, error) {
2   if b.isBusy {
3     return &pb.PubResponse{Status: pb.PubResponse_WAIT}, nil
4   }
5
6   publisher, exists := b.publishers[pub.PublisherID]
7
8   if !exists || common.CheckPublicationMAC(pub, pub.MAC, publisher.key) ==
         false {
9     return &pb.PubResponse{Status: pb.PubResponse_BAD_MAC}, nil
10   }
11
12   if b.alpha > 0 {
13     if b.alphaCounters[pub.PublisherID] == nil {
14       b.alphaCounters[pub.PublisherID] = make(map[uint64]uint64)
15     }
16
17     if pub.PubType == common.BRB {
18       b.fromPublisherCh <- *pub
19       b.alphaCounters[pub.PublisherID][pub.TopicID] = 0
20     } else {
21       if b.alphaCounters[pub.PublisherID][pub.TopicID] >= 2*b.alpha {
22         return &pb.PubResponse{Status: pb.PubResponse_BLOCKED}, nil
23       }
24
25       b.fromPublisherCh <- *pub
26
27       b.alphaCounters[pub.PublisherID][pub.TopicID]++
28       if b.alphaCounters[pub.PublisherID][pub.TopicID] == b.alpha {
29         return &pb.PubResponse{Status: pb.PubResponse_HISTORY}, nil
30       }
```

```
31      }
32    } else {
33      b.fromPublisherCh <- *pub
34    }
35
36    return &pb.PubResponse{Status: pb.PubResponse_OK}, nil
37  }
```

When a publisher publishes a publication, the broker sends back a response very quickly, because it does not need to wait for the publication to be completely processed by all the brokers and reach all the subscribers. Therefore it is possible for publishers to flood the brokers with publications to the point that brokers become overwhelmed. To overcome this, a $isBusy$ variable was added. This variable is set to true if any of the channel buffers are in danger of becoming full, and it is set to false after a timer expires and the channel buffers are at lower levels. If a publisher tries to publish something while $isBusy$ is set to true, the broker will immediately tell the publisher to wait and try again (Lines 2-4). This will help prevent the brokers from being overloaded by giving them time to process some of the publications that have accumulated in the buffers.

If the broker is not busy, it will check if the MAC is valid for publication it just received (Lines 8-10). The broker will then check if it is using a combination of two algorithms (Line 12). This is indicated by an α value greater than zero. If it using a combination, the broker will need to check if the publication is a history, or BRB, publication or not (Line 17). If it is a BRB publication, the broker will place the publication on a channel to be processed by the message handling function (Line 18). This function is described in Section 6.3. Then the alpha counter for the publisher and topic is reset.

If the publication is not a history publication, the broker will check if the α counter for that publisher and topic is greater than $2 * \alpha$ (Line 21). If the counter is greater, then the broker will tell the publisher that it will not accept any more publications with that topic until it receives a BRB history publication. This will force the publishers to send a history if it wants to continue publishing that topic. If the counter is less than $2 * \alpha$, the broker will place the publication on the channel and increment the counter. When the counter reaches α, the broker sends a response back to the publisher with the $Status$ field set to request a history publication (Line 29).

If the broker is not using a combination of algorithms, it just puts the publication on the channel. Finally if the end of the function is reached, the broker returns a response saying everything went well.

6.3 Message Handling

This section describes the message handling function of the brokers. This is the main goroutine (thread) for the brokers. Whenever an incoming message arrives from a publisher, subscriber, or broker, it is placed into a channel to be used by this routine. The for loop runs continuously throughout the life of the process. Once a message is placed into one of the channels, such as $b.fromPublisherCh$, the appropriate function to process that message is called. Some of these functions are described in the next few sections.

Listing 6.4: Message handling: Broker

```
1 func (b Broker) handleMessages() {
2   for {
3     select {
4     case req := <-b.fromPublisherCh:
5       if req.PubType == common.AB {
6         b.handleAbPublish(&req)
7       } else if req.PubType == common.BRB {
8         b.handleBrbPublish(&req)
9       } else if req.PubType == common.Chain {
10        b.handleChainPublish(&req)
11      }
12    case req := <-b.fromBrokerEchoCh:
13      b.handleEcho(&req)
14    case req := <-b.fromBrokerReadyCh:
15      b.handleReady(&req)
16    case req := <-b.fromBrokerChainCh:
17      b.handleChainForward(&req)
18    case req := <-b.fromSubscriberCh:
19      b.handleSubscribe(&req)
20    }
21  }
22 }
```

6.4 Algorithms

The following sections describe interesting pieces of code for each of the three main broadcast algorithms.

6.4.1 Authenticated Broadcast

This section describes the implementation of the Authenticated Broadcast algorithm. The code follows the pseudocode from Algorithm 2 closely. Listing 6.5 shows the code for AB. It checks if the publication has been forwarded yet. If the publication has not been forwarded, the code proceeds to forward the publication to all the subscribers. It then marks the publication as sent.

Listing 6.5: Authenticated Broadcast: Broker

```
1 func (b Broker) handleAbPublish(pub *pb.Publication) {
2   if b.forwardSent[pub.PublisherID] == nil {
3     b.forwardSent[pub.PublisherID] = make(map[int64]bool)
4   }
5
6   if b.forwardSent[pub.PublisherID][pub.PublicationID] == false {
7     pub.BrokerID = b.localID
8
9     b.subscribersMutex.RLock()
10    for _, subscriber := range b.subscribers {
11      if subscriber.toCh != nil && subscriber.topics[pub.TopicID] == true
          {
12        subscriber.toCh <- *pub
13        if len(subscriber.toCh) > b.toSubscriberChLen/2 {
14          b.setBusy()
15        }
16      }
17    }
18    b.subscribersMutex.RUnlock()
19
20    b.forwardSent[pub.PublisherID][pub.PublicationID] = true
21
22    b.incrementPublicationCount()
23  }
24 }
```

There are a few additional things that are worth noting. Lines 2-4 verify that *b.ForwardSent* has been initialized before it is used. *b.ForwardSent* is a map of maps where the first key is the publisher ID, and the second key is the publication ID. It is used to keep track of which publications have been forwarded. Similar code appears in other functions but will be omitted in any future examples.

Line 7 adds the broker's ID to the publication. This is necessary because the subscribers need to know which broker the publication comes from. If a subscriber receives three publications from the same broker, this would not enable it to reach a quorum. On the other hand, if a subscriber receives three publications from three different brokers, this would allow a quorum to be reached depending on the number of brokers.

Line 11 checks that a channel to a subscriber's write goroutine exists and that the subscriber is interested in that topic. Subscribers should only receive publications in which they are interested. Line 12 adds the publication to the channel. There is a read lock around the subscribers because the map of subscribers could change. Lines 13-15 check that the channel buffer is not becoming too full. If it is filling up, the *setBusy* function is called to set the *isBusy* flag to true and start the timer.

The *incrementPublicationCount* function on line 22 is not part of the Authenticated Broadcast algorithm. It is called to help with determining

the throughput.

6.4.2 Bracha's Reliable Broadcast

This section describes the implementation of the Bracha's Reliable Broadcast algorithm. Like the code for AB, the code for BRB closely follows its algorithms. See Algorithm 5. Listing 6.6 shows the more interesting parts of the *handleEcho*() function. If the broker has not received an echo for that publication from that broker, it proceeds to process it (Line 2). Then it marks it as received. Line 5 calls a function to check if a quorum has been reached. If the quorum has been reached and a ready message has not been sent yet for this publication, the algorithm then proceeds to send ready messages to the brokers and to interested subscribers (Lines 7-17). The actual code for sending the messages is similar to that in AB. Therefore some of the code is represented as comments and ellipses. Echo messages are the first ones that brokers broadcast in BRB, and ready messages are the second ones they broadcast.

Listing 6.6: Handle echo: Broker

```
1  func (b Broker) handleEcho(pub *pb.Publication) {
2    if b.echoesReceived[pub.PublisherID][pub.PublicationID][pub.BrokerID] ==
         "" {
3      b.echoesReceived[pub.PublisherID][pub.PublicationID][pub.BrokerID] =
           common.GetInfo(pub)
4
5      foundQuorum := b.checkEchoQuorum(pub.PublisherID, pub.PublicationID)
6
7      if foundQuorum && b.readiesSent[pub.PublisherID][pub.PublicationID] ==
           false {
8
9        // Add ready messages to all brokers' "To" channel
10       ...
11       // Add ready messages to "To" channel of all subscribers interested
             in that topic
12       ...
13
14       b.readiesSent[pub.PublisherID][pub.PublicationID] = true
15
16       b.incrementPublicationCount()
17     }
18   }
19 }
```

BRB needs to keep a record of the echo messages and the ready messages it receives. These are more nested maps. The first key is the publisher ID, the second key is the publication ID, and the third key is the broker ID. This will allow the broker to quickly check whether or not the message has been previously received. The value of these nested maps is a combination of the

topic ID and the content of the message. In addition to altering the content of a message, a malicious broker could alter the topic so that subscribers do not get messages in which they are interested or get messages in which they are not interested. Therefore it is necessary to store both. The $GetInfo()$ function will return the combination of topic and content as a string. This string will be useful in the quorum functions.

Listing 6.7 gives an example of one of the quorum functions. It takes the publisher ID and publication ID as input, and it returns whether or not a quorum exists. Inside the function there is a map which counts the number of times the specific combination of topic ID and content appears for that publisher and publication. The combination is used as the key of the map, and the counter is the value. There is a for loop that iterates over the messages received for the publisher and publication keeping a record of how many times the topic and content is seen. If the quorum is reached, the function returns true.

Listing 6.7: Check Echo Quorum

```
1  func (b *Broker) checkEchoQuorum(publisherID uint64, publicationID int64)
       bool {
2    countMap := make(map[string]uint64)
3
4    for _, echoContent := range b.echoesReceived[publisherID][publicationID]
         {
5      countMap[echoContent] = countMap[echoContent] + 1
6      if countMap[echoContent] >= b.echoQuorumSize {
7        return true
8      }
9    }
10
11   return false
12 }
```

6.4.3 Chain

This section describes the implementation of the Chain algorithm. It was a bit more challenging than initially anticipated, because the nodes in the chain could possibly have multiple parents and multiple children. Additionally the code was written so that the number of MACs checked can be changed depending on the number of replicas.

Each broker replica, publisher, or subscriber has a tree made up of chain nodes. A chain node is described in Listing 6.8. It contains the private key shared between the local node and the tree node. It also contains IDs of the children and the parents of the tree node. These IDs are strings which are a combination of the node type and the numerical ID of the node. For

example, publisher 1 would have the ID string "P1". Since there could be nodes of different types with the same numerical ID, e.g. publisher 1 and broker 1, there needed to be a way to differentiate between them. All the nodes are stored in a map using the ID strings as the key in the map.

Listing 6.8: Chain Node

```
1 type chainNode struct {
2   key      []byte
3   children []string
4   parents  []string
5 }
```

Listing 6.9 shows how a broker node handles receiving a chain publication. Only the interesting parts are shown to reduce the size. When handling a Chain publication, the *handleChainPublish* function calls the *verifyChainMACs* function to verify that the MACs between this node and all of its ancestors up to the range of the chain are correct. Then for each of the child nodes of the local node, a new publication is created. Most of the values will be the same as the publication received, but the broker's ID and the chain MACs will be different. Each child will receive a different set of chain MACs. Therefore each child needs its own publication. Next the function calls the *addMACsRecursive* function to add all the MACs required by that child and the child's descendants. Once the MACs are added, the publication is ready to be sent to the child and is added to the child's channel for output.

Listing 6.9: Handle Chain Publications: Broker

```
1  func (b *Broker) handleChainPublish(pub *pb.Publication) bool {
2    if b.chainSent[pub.PublisherID][pub.PublicationID] == true {
3      return false
4    }
5
6    verified := b.verifyChainMACs(pub, b.chainRange)
7    if !verified {
8      return false
9    }
10
11   for _, childStr := range b.chainNodes[b.localStr].children {
12     tempPub := &pb.Publication{
13       PubType:       pub.PubType,
14       PublisherID:   pub.PublisherID,
15       PublicationID: pub.PublicationID,
16       TopicID:       pub.TopicID,
17       BrokerID:      b.localID,
18       Contents:      pub.Contents,
19     }
20
21     b.addMACsRecursive(tempPub, pub, childStr, b.chainRange)
22
23     // Add the publication to the child's "To" channel
```

```
24      ...
25    }
26
27    b.chainSent[pub.PublisherID][pub.PublicationID] = true
28
29    b.incrementPublicationCount()
30
31    return true
32  }
```

The *addMACsRecursive* function will recursively add MACs necessary for a child node and all of its descendants. This is shown in Listing 6.10. To avoid any confusion, "current node" will refer to a node that is changed as the recursion moves through the tree, and "local node" will refer to the node which is running the process. First the function will search through the MACs from the previous message, which is the message from the previous node to the local node, to see if there are any pertinent MACs for the current node in the tree. Looking back at Figure 5.5, the green MACs from the old message can become the pink MACs in the new message if they are pertinent to the destination of the new message. If any are found, they will be added to the new message's MACs.

Instead of just adding all the old MACs to the slice of MACs each time a message is forwarded, it was decided to only send the MACs that are needed in the future. MACs that are no longer required are not added to the forwarded messages. This helps keep the slice of MACs from growing too large.

Next the function will check how many generations of MACs are left to add. It might seem a little strange that it is checking that the number of generations is greater than one as opposed to greater than zero. The MAC for the local node and its child is handled the exact same way for all the algorithms. Therefore it is unnecessary to add the same MAC twice, and the *addMACsRecursive* function only needs to go through one less than the chain range.

Once the generation check is passed, the function proceeds to add a MAC for each child of the current node. Additionally if the child is a subscriber, it must be interested in the publication's topic in order for a MAC to be added (Lines 15-17). It would waste resources to calculate MACs for subscribers who are not subscribed to that topic. The *From* location is the local node and the *To* location is the child of the current node. Finally the *addMACsRecursive* function will be called on that child.

Listing 6.10: Recursively Add MACs

```
1 func (b *Broker) addMACsRecursive(pub *pb.Publication, oldPub *pb.
    Publication, currentStr string, generations int) {
2   for _, oldChainMAC := range oldPub.ChainMACs {
3     if oldChainMAC.To == currentStr {
4       pub.ChainMACs = append(pub.ChainMACs, oldChainMAC)
5     }
6   }
7
8   if generations > 1 {
9     for _, childStr := range b.chainNodes[currentStr].children {
10      addMAC := true
11
12      if strings.HasPrefix(childStr, "S") {
13        childID, _ := strconv.ParseUint(childStr[1:], 10, 64)
14        b.subscribersMutex.RLock()
15        if b.subscribers[childID].topics[pub.TopicID] == false {
16          addMAC = false
17        }
18        b.subscribersMutex.RUnlock()
19      }
20
21      if addMAC {
22        chainMAC := pb.ChainMAC{
23          From: b.localStr,
24          To:   childStr,
25          MAC:  common.CreatePublicationMAC(pub, b.chainNodes[childStr].
              key),
26        }
27
28        pub.ChainMACs = append(pub.ChainMACs, &chainMAC)
29
30        b.addMACsRecursive(pub, oldPub, childStr, generations-1)
31      }
32    }
33  }
34 }
```

6.5 Channels and Pointer References

The channels used in passing messages between different threads in the same process are described in this section. Originally it was planned to just add pointers to the messages on the channels, thereby passing by reference. This would have saved some processing time because less data would have to be copied over into new variables. Unfortunately this caused some issues when a message was sent to multiple receivers. Each receiver needs its own MAC, and changing the MAC several times on the same message caused some receivers to get incorrect MACs. This error was not reproducible every time and took a bit of debugging to find. The messages themselves were added to the channels to fix this, thereby passing by value.

6.6 MACs

This section describes how the MACs are created for the publications. Listing 6.11 shows the $CreatePublicationMAC()$ function that is used for all publications regardless of their publication type to ensure a fair comparison between the types. It takes a publication and a private key as input parameters and returns a byte slice containing the publication's MAC. It will work with the MD5, SHA1, SHA256, and SHA512 hash algorithms. The algorithm is stored in the *Algorithm* variable. The *hash* and *crypto/hmac* packages in Go do all the major work in calculating the MACs.

Listing 6.11: Create publication MAC

```
1 func CreatePublicationMAC(pub *pb.Publication, key []byte) []byte {
2   var mac hash.Hash
3   message := ConvertPublicationToBytes(pub)
4
5   switch Algorithm {
6   case crypto.MD5:
7     mac = hmac.New(md5.New, key)
8   case crypto.SHA1:
9     mac = hmac.New(sha1.New, key)
10  case crypto.SHA256:
11    mac = hmac.New(sha256.New, key)
12  case crypto.SHA512:
13    mac = hmac.New(sha512.New, key)
14  default:
15    mac = hmac.New(md5.New, key)
16  }
17  mac.Write(message)
18  return mac.Sum(nil)
19 }
```

Listing 6.12 shows the $ConvertPublicationToBytes()$ function, which will take the relevant information from a publication and convert it to a slice of bytes. Several of the fields, such as $PublisherID$ and $PublicationID$, in the publication are converted to byte slices (lines 8-11). The $BrokerID$ field is not used, because it changes with every broker it passes through. Normally this would be added, but it causes problems for the Chain algorithm. If $BrokerID$ was used, then any MACs that are not between parent and child would be invalid.

Once the numerical fields have been converted to bytes, they are written to a buffer. Then each of the messages in the *Contents* field is written to the buffer. Most publications will just have one message, but the history publications will have several. Finally the function converts the buffer to a slice of bytes and returns it.

Listing 6.12: Converting the Publication to a byte slice

```
1  func ConvertPublicationToBytes(pub *pb.Publication) []byte {
2    var buf bytes.Buffer
3    pubType := make([]byte, 8)
4    publisherID := make([]byte, 8)
5    publicationID := make([]byte, 8)
6    topicID := make([]byte, 8)
7
8    binary.PutUvarint(pubType, uint64(pub.PubType))
9    binary.PutUvarint(publisherID, pub.PublisherID)
10   binary.PutVarint(publicationID, pub.PublicationID)
11   binary.PutUvarint(topicID, pub.TopicID)
12
13   buf.Write(pubType)
14   buf.Write(publisherID)
15   buf.Write(publicationID)
16   buf.Write(topicID)
17   for i := range pub.Contents {
18     buf.Write(pub.Contents[i])
19   }
20
21   return buf.Bytes()
22 }
```

The code for validating a MAC is very simple and is shown in Listing 6.13. The function $CheckPublicationMAC()$ takes the publication, the MAC calculated by the sender, and the private key as input parameters. It then creates a MAC by itself, the receiver, using the $CreatePublicationMAC()$ function described earlier. Then it returns whether or not the MAC from the sender and the MAC from the receiver are equal.

Listing 6.13: Validating a MAC

```
1 func CheckPublicationMAC(pub *pb.Publication, mac []byte, key []byte) bool
      {
2   mac2 := CreatePublicationMAC(pub, key, Algorithm)
3   return hmac.Equal(mac, mac2)
4 }
```

6.7 Malicious Broker

This section describes the code which purposely causes a broker to behave maliciously. In addition to testing performance in a normal scenario, it was decided to test it when a broker replica is misbehaving. If a command line argument is set to greater than 100 when the broker is started, the broker will not send any publications to subscribers, any BRB messages to f brokers, or any Chain messages to the next node in the chain. Recall that f is the number of faults that can be tolerated given the total number of brokers. Along with a publisher that ignores one broker, this will enable the error

scenarios described in Chapter 2. Then publications sent with the AB and Chain algorithms will not reach the subscribers, but publications sent with the BRB algorithm will reach them.

If the command line argument is set to a value between 1 and 100, the malicious broker will alter that percentage of publications. The function shown in Listing 6.14 is called to possibly alter one of the fields in the publication just before the MAC is calculated. A number from 1 to 100 is randomly generated. If that number is less than or equal to the percent of publications to alter, one of the fields is changed. The field that will be altered is randomly chosen.

Listing 6.14: Maliciously alter publication

```
1  func (b *Broker) alterPublication(pub *pb.Publication) pb.Publication {
2    tempPub := pb.Publication{
3      PubType:       pub.PubType,
4      PublisherID:   pub.PublisherID,
5      PublicationID: pub.PublicationID,
6      TopicID:       pub.TopicID,
7      BrokerID:      pub.BrokerID,
8    }
9
10   for i := range pub.Contents {
11     tempPub.Contents = append(tempPub.Contents, pub.Contents[i])
12   }
13   for i := range pub.ChainMACs {
14     tempPub.ChainMACs = append(tempPub.ChainMACs, pub.ChainMACs[i])
15   }
16
17   r := b.random.Intn(101)
18
19   if r <= b.maliciousPercent {
20     var alterType int
21
22     if len(tempPub.ChainMACs) > 0 {
23       alterType = r % 6
24     } else {
25       alterType = r % 5
26     }
27     switch alterType {
28     case 0:
29       tempPub.PublicationID = tempPub.PublicationID + 1
30     case 1:
31       tempPub.PublisherID = tempPub.PublisherID + 1
32     case 2:
33       tempPub.BrokerID = tempPub.BrokerID + 1
34     case 3:
35       tempPub.TopicID = tempPub.TopicID + 1
36     case 4:
37       if len(tempPub.Contents) > 0 {
38         tempPub.Contents[0] = badMessage
39       }
40     case 5:
41       tempPub.ChainMACs = nil
42     }
```

```
43   }
44
45   return tempPub
46 }
```

Chapter 7

Results

This chapter discusses the test scenarios and shows their results. The throughput and the latency of the algorithms and combinations of the algorithms were tested. Section 7.1 describes how the tests were run and the results calculated. Section 7.2 describes the testing environments. Section 7.3 provides information on the tests and the results obtained from those tests.

7.1 Test Setup

This section describes in general how the tests were setup and run. Since the majority of the processing of the algorithms is done by the brokers, it was necessary for each broker replica was placed on its own machine. The publishers and subscribers do much less processing. Therefore it was possible to place multiple publishers and subscribers on a machine. Four machines were used for up to a total of fourteen publishers and fourteen subscribers.

Publishers generated new publications as fast as they were able to so that they could push the brokers to their capacity. Each publisher sent 50,000 publications during each test, not including history publications.

The length of the content, or payload, of the publications in the tests was 56 bytes. This does not include protobuf information, MACs, publisher ID, publication ID, etc. The hash function used was MD5.

Latency is the time it takes for a publication to be sent from a publisher to a subscriber. When a publisher sent a publication, it recorded the publisher ID, the publication ID, and the time to a file. When a subscriber received a publication, it recorded the same information along with its own ID to another file. Once the test was over, the files were compared. The sent time was subtracted from the received time for publications that matched the

publisher ID and publication ID. This produced the latencies and various statistics were generated from these. The history publications were included in the latency.

Throughput is the number of publications that the brokers could process during a certain amount of time. At the final step of each algorithm, a broker incremented a counter signifying the number of publications that it had processed. After every second, that number was recorded, and the counter was reset. When calculating the statistics on the throughput, the zeros recorded before a test began and after a test finished were ignored. The history publications were not calculated in the throughput, unless they contained publications that were missed due to arbitrary failures.

7.2 Testing Environment

This section describes the testing environment. A group of computers located in one of the labs at the University of Stavanger was used. Each machine used had the following configuration.

Table 7.1: UiS Computer Configuration

Number of Processors	4
Type of Processor	Intel(R) Xeon(R) CPU E5606 @ 2.13GHz
Memory	4 GB
Network Adapter	1 Gigabit
OS	CentOS

7.3 Tests

This section describes the specific tests that were run. First a series of baseline tests were run with four brokers. Then other tests were run by adding more brokers, adding more topics, or making one of the brokers faulty.

7.3.1 Four Broker Replicas

In this section, we list the results of a series of tests with four broker replicas. These tests form a baseline that the other tests are compared against.

The number of publishers and subscribers were increased until the brokers reached their maximum throughput. Once it was apparent that the

throughput had reached a plateau or a peak for a specific algorithm, the tests on that algorithm were stopped. For example, the throughput for BRB did not increase significantly with one through four publishers, so there was no point in increasing them further in the BRB tests. Figure 7.1 shows the throughput and Figure 7.2 shows the latency. The throughput is measured in publications per second (pps), and the latency is measured in milliseconds (ms).

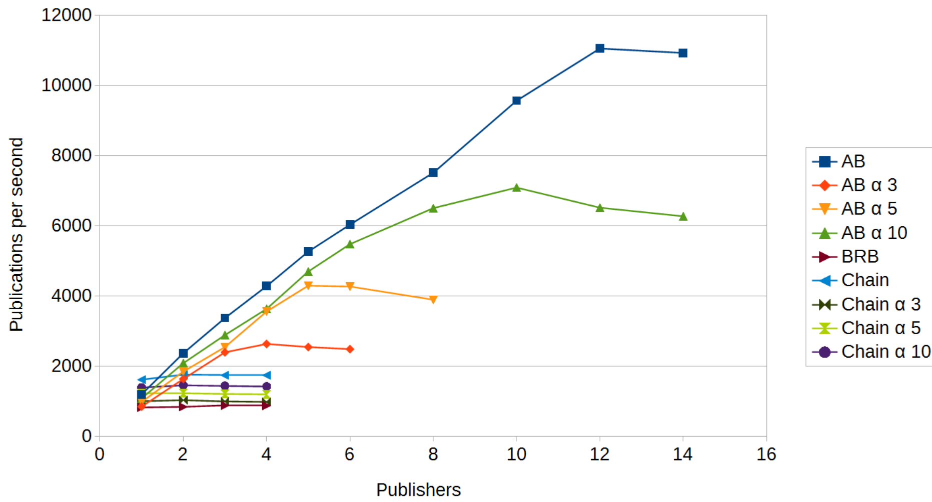

Figure 7.1: Throughput with 4 brokers.

The throughput with AB was very good with a peak around 11,000 pps. The number of publishers needed to be increased to twelve to reach a plateau. BRB and Chain reached a plateau with just one publisher sending publications as fast as it could. Their throughputs were about 850 and 1750 respectively. The combinations of algorithms were able to achieve throughputs that were between the basic algorithms. With higher α values, the combination algorithms were able to get better throughput. It was expected that the throughput of Chain would be close to AB, but instead it was closer to BRB.

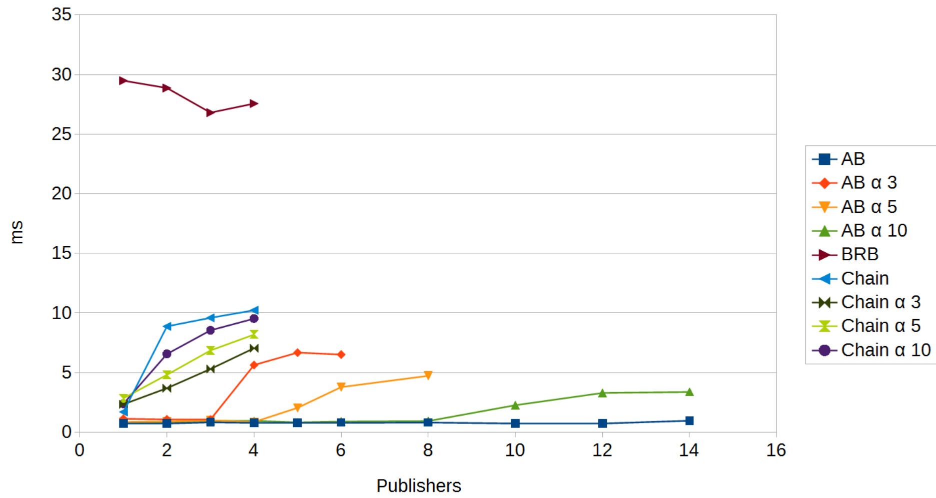

Figure 7.2: Latency with 4 brokers.

BRB had the worst latencies with values near 30 ms. This is probably caused by the number of messages required by BRB and the number of quorums needed. AB and Chain had lower latencies. It is interesting that the latencies for AB and the different combinations of AB and BRB had latencies of one ms or less up to a certain point. When the number of BRB publications began to saturate the replicas, the latencies began to increase. When the α value was increased, the point at which the latencies increased moved further down the graph.

Figures 7.3-7.7 show the distribution of latencies for selected algorithms when they reached their maximum throughput. For example, AB reached its peak with twelve publishers and AB with α 10 reached its peak with ten publishers. Each publisher sent 50,000 publications. If there is an α value, the latencies of the history publications are included. The blue lines are the mean, the green lines are the standard deviation, and the red lines are the 99th percentile. The graphs were scaled to fit all the outliers. The latencies are not in chronological order. They are in a random order from iterating over a map in Go.

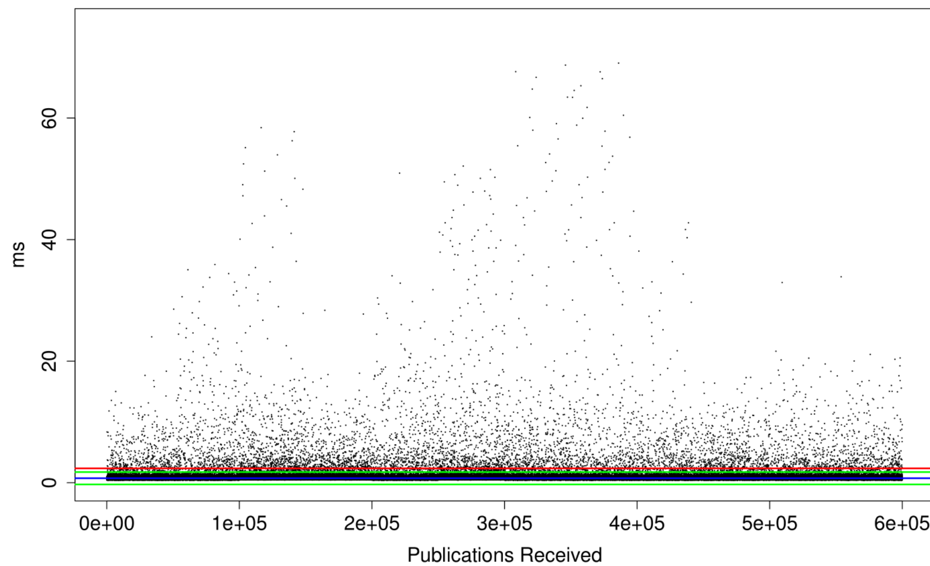

Figure 7.3: Latencies for AB: 12 publishers

The latencies for AB were mostly very low. The mean was 0.7 ms, the standard deviation was 1.0 ms, and the 99$^{\text{th}}$ percentile was 2.4 ms. 600,000 AB publications were sent in this test.

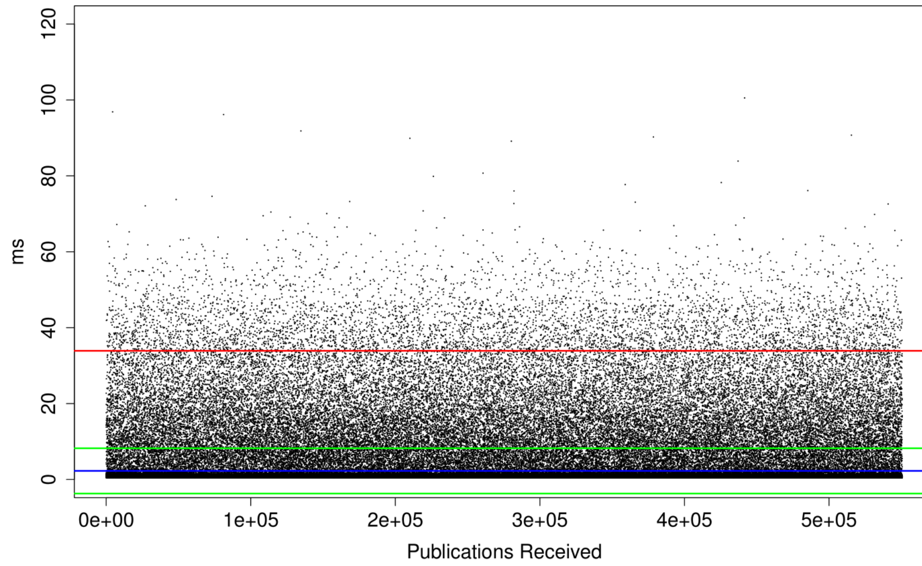

Figure 7.4: Latencies for AB with α 10: 10 publishers

The latencies for AB with α 10 showed more variation. The mean was 2.3 ms, the standard deviation was 6.0 ms, and the 99$^{\text{th}}$ percentile was 33.9 ms. This was caused by the large number of BRB publications sent. 500,000 AB publications and 50,000 BRB publications were sent in this test.

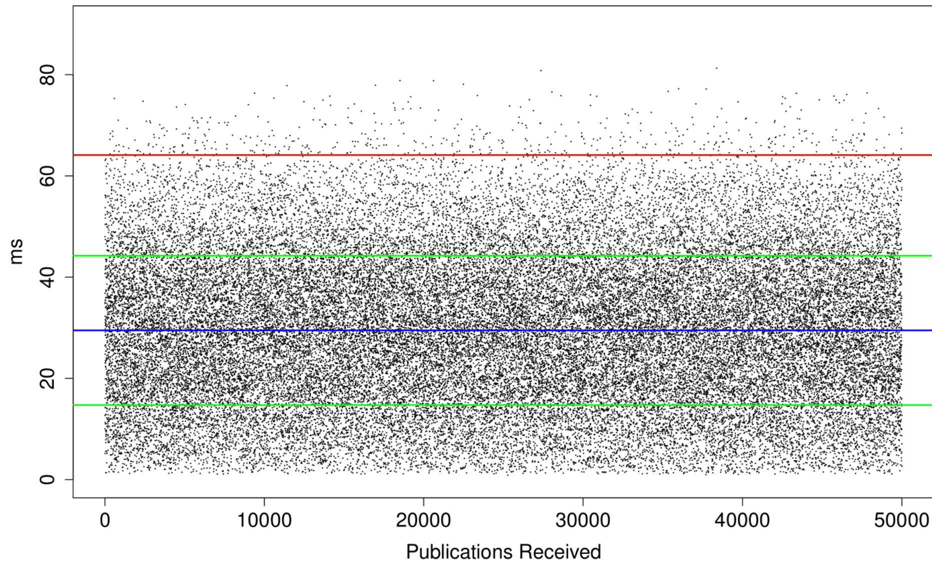

Figure 7.5: Latencies for BRB: 1 publisher

The latencies for BRB had the most variation. The mean was 29.5 ms, the standard deviation was 14.7 ms, and the 99$^{\text{th}}$ percentile was 64.1 ms. The amount of variation was due to the high number of messages sent in this algorithm. A subscriber had to wait for the brokers to reach a quorum before it could reach a quorum itself. 50,000 BRB publications were sent in this test.

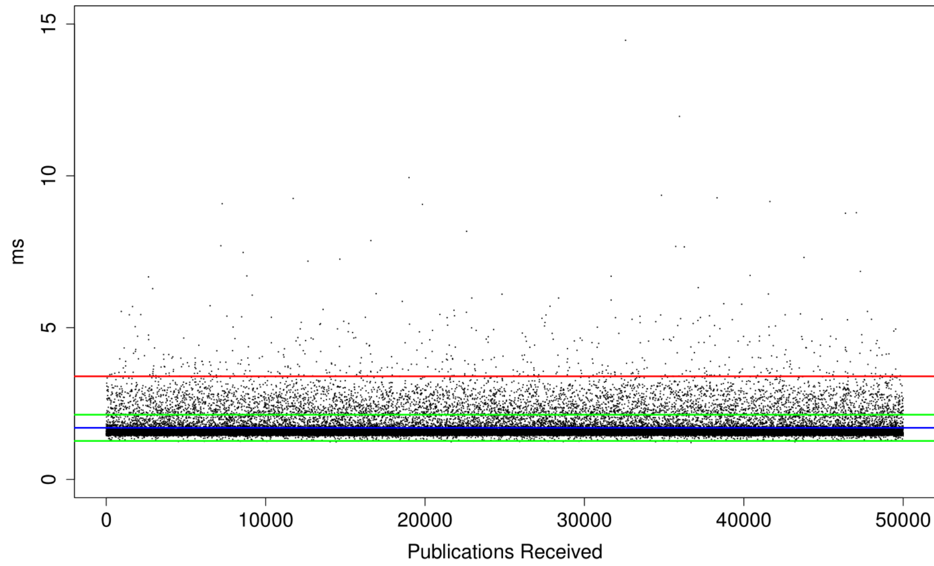

Figure 7.6: Latencies for Chain: 1 publisher

The latencies for Chain were low. The mean was 1.7 ms, the standard deviation was 0.4 ms, and the 99$^{\text{th}}$ percentile was 3.4 ms. 50,000 Chain publications were sent in this test. The latencies in Chain were higher than the latencies in AB, because the publications in Chain went through more communication steps. While AB uses more messages than Chain, it only has two communications steps compared to four communication steps in Chain.

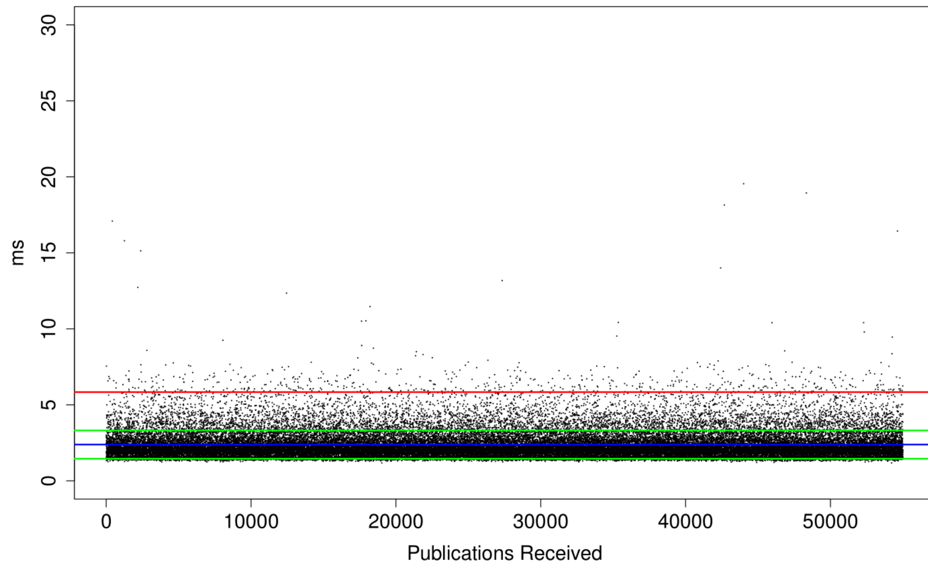

Figure 7.7: Latencies for Chain with α 10: 1 publisher

The latencies for AB with α 10 were low also. The mean was 2.4 ms, the standard deviation was 0.9 ms, and the 99$^{\text{th}}$ percentile was 5.8 ms. 50,000 Chain publications and 5,000 BRB publications were sent in this test. This test had less BRB messages than the AB with α 10 test. Therefore it had less variation.

7.3.2 Seven Broker Replicas

This section shows the results of a series of tests with seven broker replicas. The throughput is shown in Figure 7.8. This graph looks very similar to the throughput graph for four brokers, Figure 7.1. The throughputs for BRB, Chain, and their combinations reach a plateau immediately, while the throughputs for AB, and the combinations of AB and BRB steadily increase to eventually come to a peak or plateau. AB got a max throughput of 7400 pps, which is significantly less than what it got with only four brokers. The throughputs of BRB and Chain were closer to the their results from the baseline test.

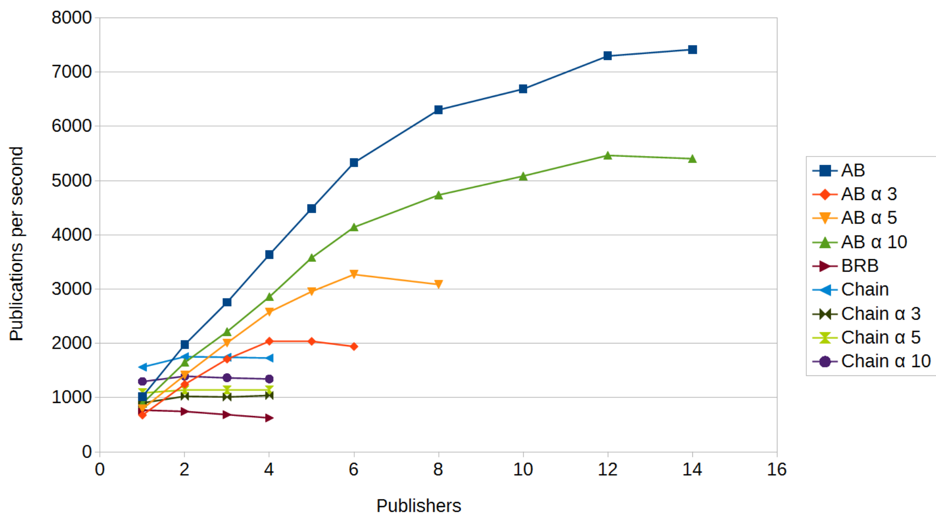

Figure 7.8: Throughput with 7 brokers.

Figure 7.9 shows the latencies for the seven broker test. Generally this graph looks similar to the latency graph for four brokers, Figure 7.2. The latencies for the combinations of AB and BRB started off around one ms and then increased once the replicas started to become saturated. The latencies for the combinations of Chain and BRB seem to gradually increase. The latency for BRB seems to be a little usual. When there was only one publisher, it had a much shorter latency than in the four broker test. Overall the latencies were longer with seven brokers than with four brokers.

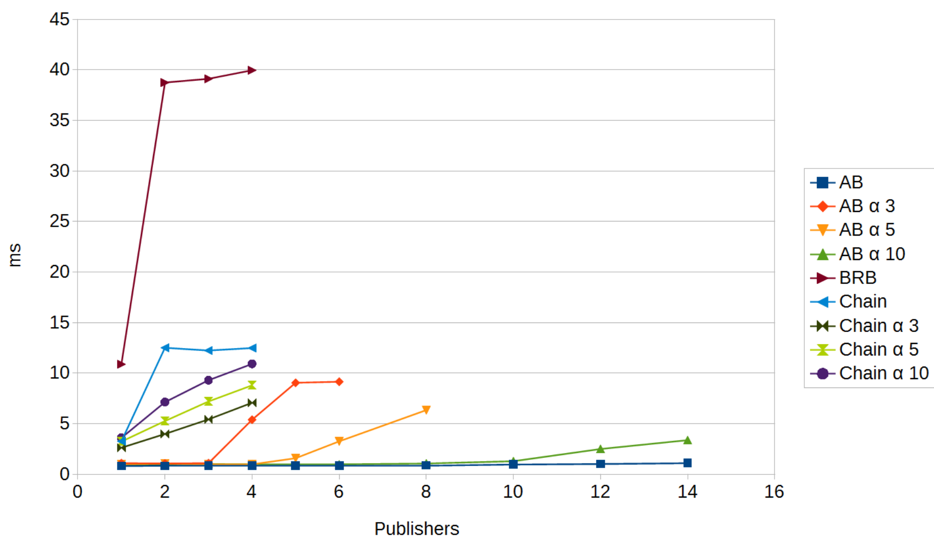

Figure 7.9: Latency with 7 brokers.

Figure 7.10 compares the maximum throughputs for each algorithm from the four and seven broker tests. In this graph, it is much easier to see the difference between them. Increasing the number of broker replicas lowered the maximum throughput, especially for AB and combinations of AB and BRB.

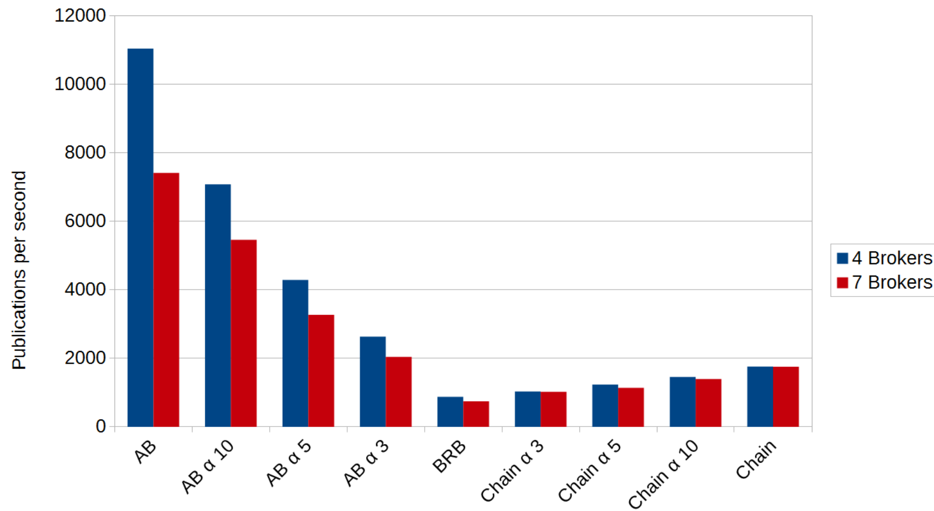

Figure 7.10: Comparing throughputs with 4 and 7 brokers.

Figure 7.11 compares latencies from the four and seven broker tests when the algorithms were at their maximum throughput. Increasing the number of broker replicas increased the latency, especially for BRB and Chain.

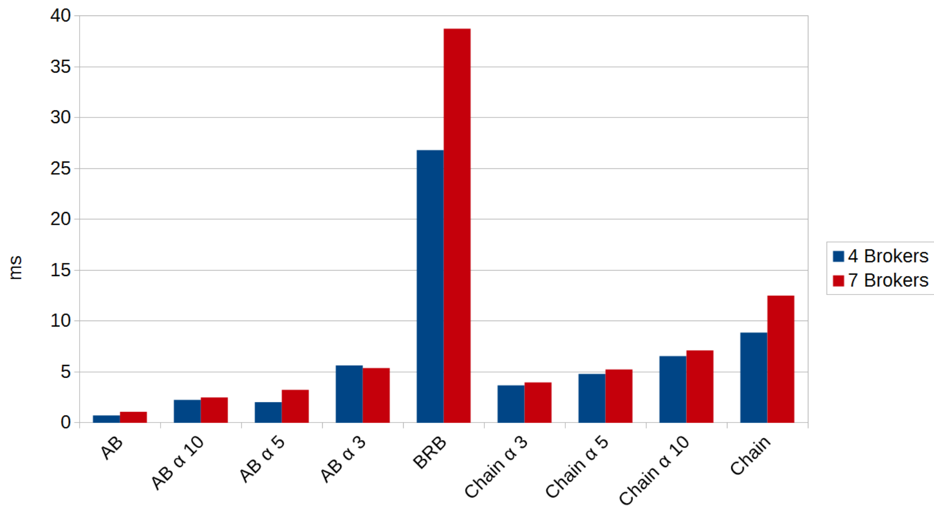

Figure 7.11: Comparing latencies with 4 and 7 brokers.

7.3.3 Multiple Topics

This section shows the results of increasing the number of topics for each publisher from one to four. Figure 7.12 compares the maximum throughputs. The throughput was similar, but the difference seems to increase in higher levels of throughput. AB had the highest throughput, and it had the largest decrease in throughput when the number of topics was changed from one to four. BRB and Chain had lower throughput, and changing the number of topics did not affect them as much.

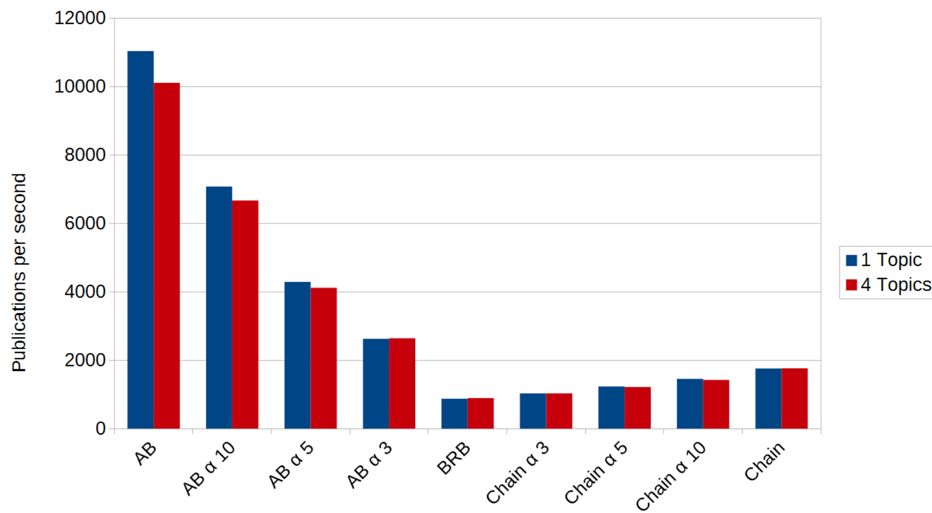

Figure 7.12: Comparing throughputs with 1 and 4 topics.

Figure 7.13 compares the latency when the maximum throughput is reached. The latency in the basic algorithms did not change much. However, the latency was higher when combinations of algorithms were used, which seems unusual.

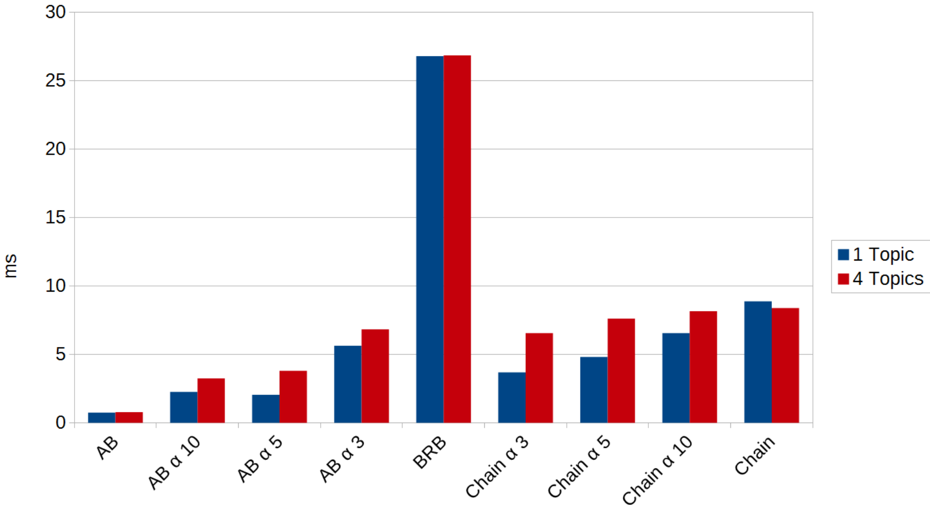

Figure 7.13: Comparing latencies with 1 and 4 topics.

7.3.4 Malicious Broker

This section gives the results of the malicious broker tests. Two series of tests were run. In the first series, the malicious broker chose not to send publications to one broker or any subscriber. In the second series, the malicious broker altered the contents of every publication to one broker and all subscribers. In addition to the malicious broker, the publishers also had to ignore one of the brokers in both series of tests. In these scenarios, the AB and Chain algorithms failed to deliver any publications to subscribers. However when they were combined with BRB, the subscribers received the publications. This was how the tests were intended to work. This shows that the history publication works correctly.

Figures 7.14 and 7.15 show the throughput for the malicious broker tests. Neither of these two graphs have lines for AB nor Chain, because both of the algorithms failed to deliver any publications to the subscribers. Otherwise they are very similar to the throughput graph from the baseline test, Figure 7.1.

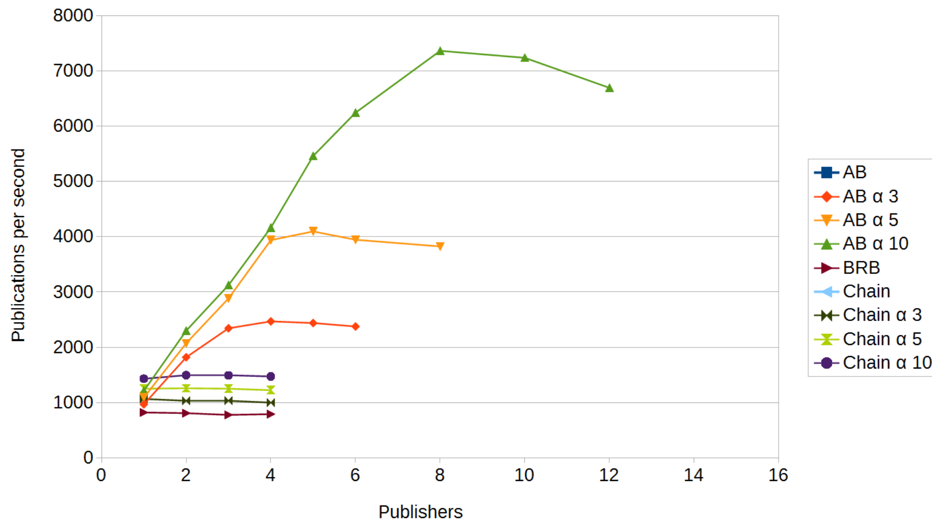

Figure 7.14: Throughput with a broker selectively dropping publications.

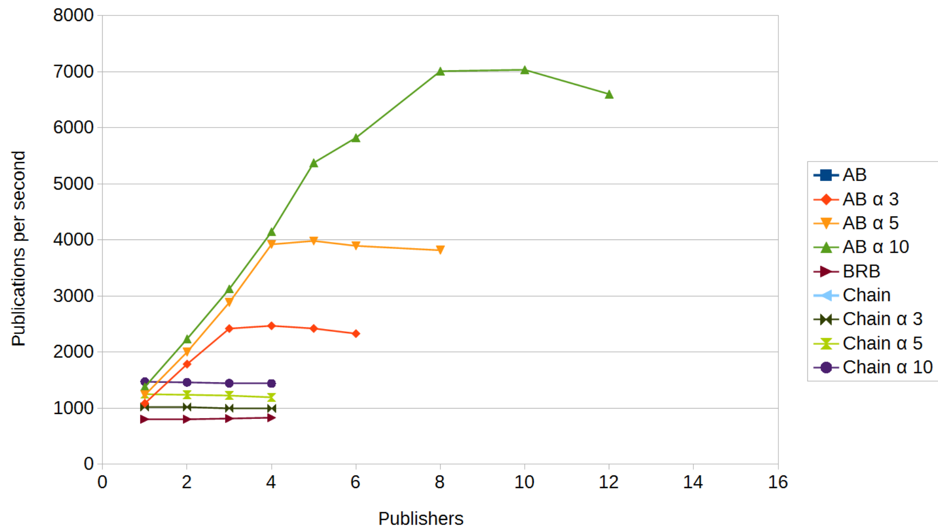

Figure 7.15: Throughput with a broker selectively altering publications.

Figures 7.16 and 7.17 show the latency for the malicious broker tests. These do not show latencies for AB or Chain. Unlike the latency graph for the baseline test, Figure 7.2, there is a significant rise in the latencies when the replicas become saturated with the combination of AB and BRB.

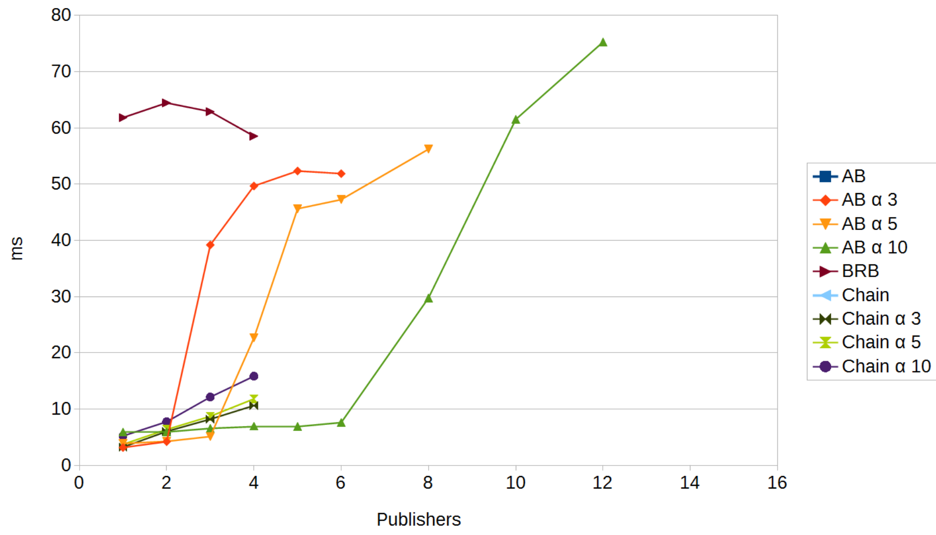

Figure 7.16: Latency with a broker selectively dropping publications.

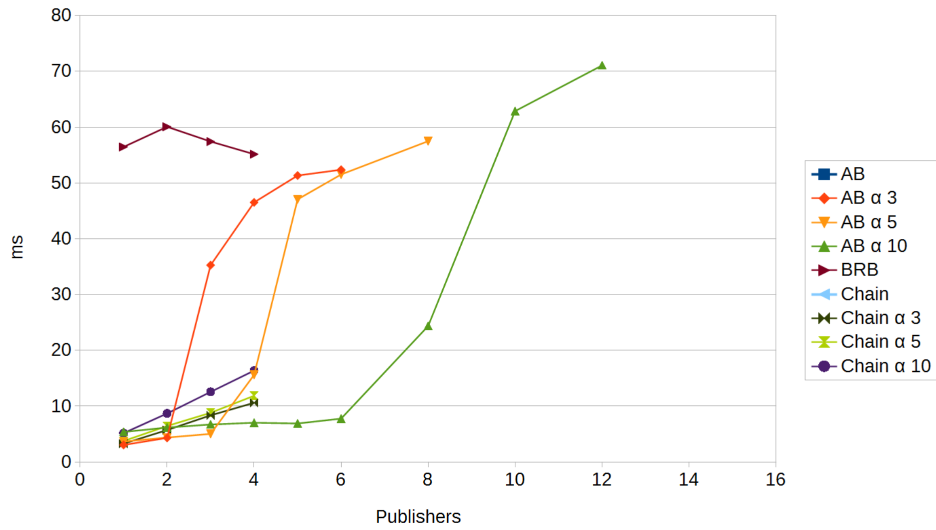

Figure 7.17: Latency with a broker selectively altering publications.

Figure 7.18 compares the maximum throughput for the error scenarios with the base scenarios. The throughput did not change much since the history publications contained the missing publications. It was expected that the throughput would not change much, because the history publications are always sent.

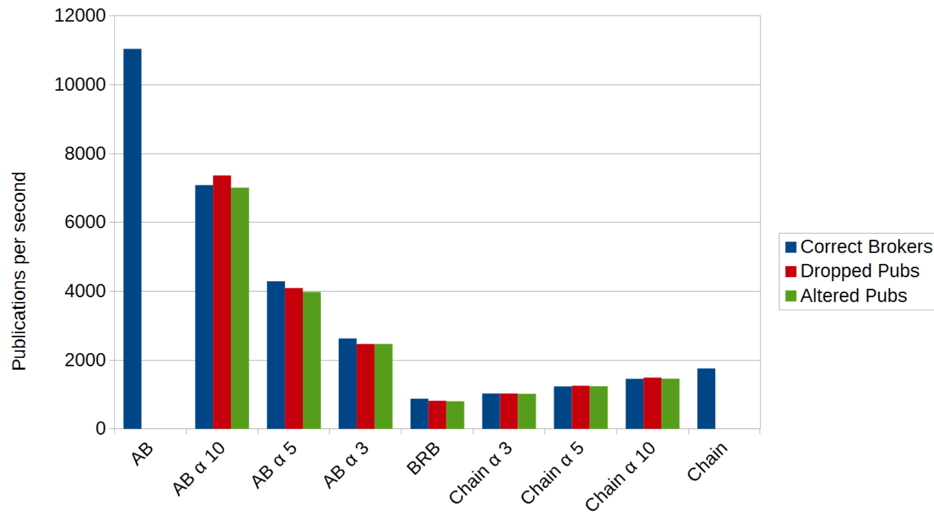

Figure 7.18: Comparing throughput with correct and malicious brokers.

However, the latencies increased in the error scenarios because the subscribers had to wait for the history publications to get all publications. Figure 7.19 compares the latency for the error scenarios with the base scenarios. The latencies in the Chain algorithm with a history did not see the dramatic rise that was seen in the AB algorithm with a history. Maybe this is due to the lower observed throughput with Chain. A lower throughput means less BRB history publications per second to affect the latency. If the throughput of Chain was higher, as it was expected to be, perhaps the latency of the combined algorithms would be higher as well. The pipelined nature of Chain might be affecting this also.

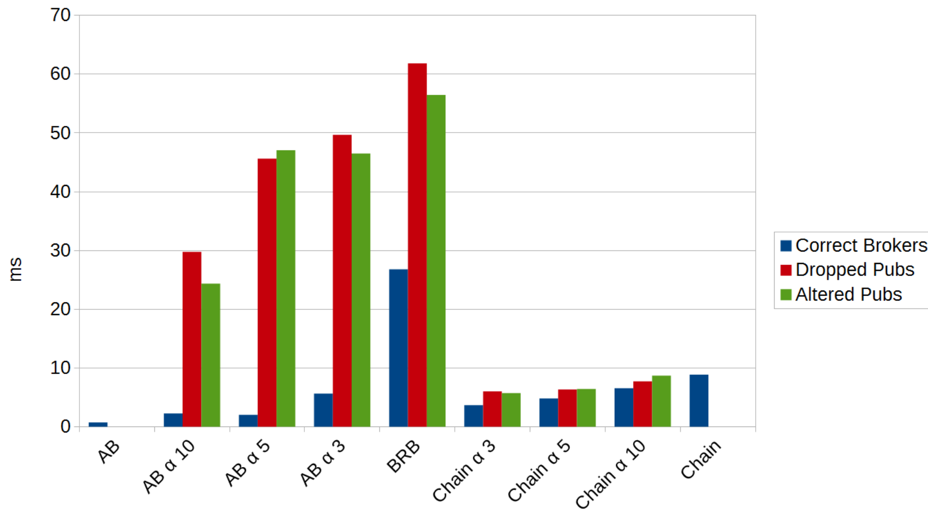

Figure 7.19: Comparing latencies with correct and malicious brokers.

Chapter 8

Future work

This chapter discusses possible future work regarding combining broadcast algorithms. Section 8.1 suggests finding the optimal α value. Section 8.2 looks at improving the performance in the implementation of Chain. Section 8.3 mentions some additional algorithms that could be added to the project, and Section 8.4 suggests implementing a tree structure.

8.1 Optimal α Value

Is there an optimal α value? The throughput seems to increase as α increases. It gets closer and closer to the throughput of the original algorithm. Will this always happen, or will the history publication become so large that the throughput starts to decrease at some point? This would be worth investigating.

8.2 Bottleneck in Chain Publications

The Chain algorithm did not perform as well as expected. Its throughput was closer to BRB than it was to AB. Some work was already done to find the bottleneck. Commenting out all the code for adding and checking MACs only improved throughput by 60 publications per second. Therefore it is probably not related to that. Perhaps the bottleneck is in the flow control method mentioned in Section 6.2.

8.3 Additional Algorithms

Additional algorithms could be added to application and combined to see how they perform. Some possible algorithms are Zyzzyva [2], BChain [25], and Practical Byzantine Fault Tolerance [3].

8.4 Tree Structure

Another idea worth pursuing is implementing the tree structure for multiple broker replicated state machines mentioned in [11]. This would allow the application to be more scalable.

Chapter 9

Conclusion

In this thesis, a few different broadcasting algorithms were implemented: Authenticated Broadcast, Bracha's Reliable Broadcast, and Chain. Later the AB and Chain algorithms were combined with BRB, the most expensive and reliable algorithm, being used to distribute a history publication. When subscribers had missed normal publications due to a faulty broker replica, they were able to learn the publications later through the history publication. The combination of algorithms allowed the throughput to be close to that of the more efficient algorithms, AB and Chain, while having the reliability of BRB.

The application followed the Publish/Subscribe pattern where publishers would send publications through an intermediary service, which would then forward them to interested subscribers. The intermediary service was made of a group of replicated state machines. All of the communication was made through gRPC remote procedure calls.

Tests were run in a number of scenarios. The α value, the number of broker replicas, and the number of topics published by a publisher were varied. Increasing α led to higher throughputs, because the more efficient algorithm was used more often. Adding more brokers to increase the reliability decreased the throughput of the brokers when using AB and BRB. It did not seem to affect Chain as much due to its pipelined nature. Adding more topics to each publisher lowered the throughput slightly.

Also tests were run to simulate a broker with Byzantine faults in such a way that caused AB and Chain algorithms to fail to deliver any publications. When those algorithms were combined with BRB, the publications were able to reach the subscribers thereby verifying the reliability of the combined algorithms. The throughput did not change much during the error scenarios.

However, the latency increased dramatically, because the subscribers needed to wait for the history publication to receive the normal publications that were lost.

In conclusion, there is a benefit from combining an efficient algorithm, such as Authenticated Broadcast or Chain, with a more reliable algorithm like Bracha's Reliable Broadcast. When the reliable algorithm is used to send history messages, one can get a service that is fast and dependable.

Bibliography

[1] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," *ACM Transactions on Computer Systems*, vol. 0, no. 0, 2014.

[2] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (New York, NY, USA), pp. 45–58, ACM, 2007.

[3] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, (Berkeley, CA, USA), pp. 173–186, USENIX Association, 1999.

[4] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," *SIGOPS Oper. Syst. Rev.*, vol. 39, pp. 59–74, Oct. 2005.

[5] J. C. Adams and K. V. S. Ramarao, "Distributed diagnosis of byzantine processors and links," in *Distributed Computing Systems, 1989., 9th International Conference on*, pp. 562–569, Jun 1989.

[6] R. Baldoni, J. M. Helary, and M. Raynal, "From crash fault-tolerance to arbitrary-fault tolerance: towards a modular approach," in *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pp. 273–282, 2000.

[7] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, (Berkeley, CA, USA), pp. 153–168, USENIX Association, 2009.

[8] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), pp. 277–290, ACM, 2009.

[9] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony (preliminary version)," in *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, (New York, NY, USA), pp. 103–118, ACM, 1984.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, pp. 374–382, Apr. 1985.

[11] L. Jehl and H. Meling, "Towards byzantine fault tolerant publish/subscribe: A state machine approach," in *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*, HotDep '13, (New York, NY, USA), pp. 5:1–5:5, ACM, 2013.

[12] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, July 1982.

[13] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Pearson Education Limited, 6th ed., 2014.

[14] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Springer, 2nd ed., 2011.

[15] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, pp. 133–169, May 1998.

[16] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," *SIGOPS Operating Systems Review*, vol. 21, pp. 123–138, Nov. 1987.

[17] "Effective go - the go programming language." `https://golang.org/doc/effective_go.html`. Accessed: 2016-01-27.

[18] "Developer guide | protocol buffers | google developers." `https://developers.google.com/protocol-buffers/docs/overview`. Accessed: 2016-01-20.

[19] "grpc / documentation." http://www.grpc.io/docs/. Accessed: 2016-01-20.

[20] T. Lea, L. Jehl, and H. Meling, "Gorums: A framework for implementing reconfigurable quorum-based systems." In submission, 2016.

[21] T. Lea, "Gorums." Presentation. Presented: 2016-01-20.

[22] T. K. Srikanth and S. Toueg, "Simulating authenticated broadcasts to derive simple fault-tolerant algorithms," *Distributed Computing*, vol. 2, pp. 80–94, June 1987.

[23] G. Bracha, "Asynchronous byzantine agreement protocols," *Information and Computation*, vol. 75, pp. 130–143, Nov. 1987.

[24] E. Haaland and S. Ramstad, "Tolerating arbitrary failures in a pub/sub system," Bachelor's thesis, University of Stavanger, May 2014.

[25] S. Duan, H. Meling, S. Peisert, and H. Zhang, *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*, ch. BChain: Byzantine Replication with High Throughput and Embedded Reconfiguration, pp. 91–106. Cham: Springer International Publishing, 2014.

Appendix A

Attachments

Source code: